AIMERZ.ai
Aim.Act.Achieve

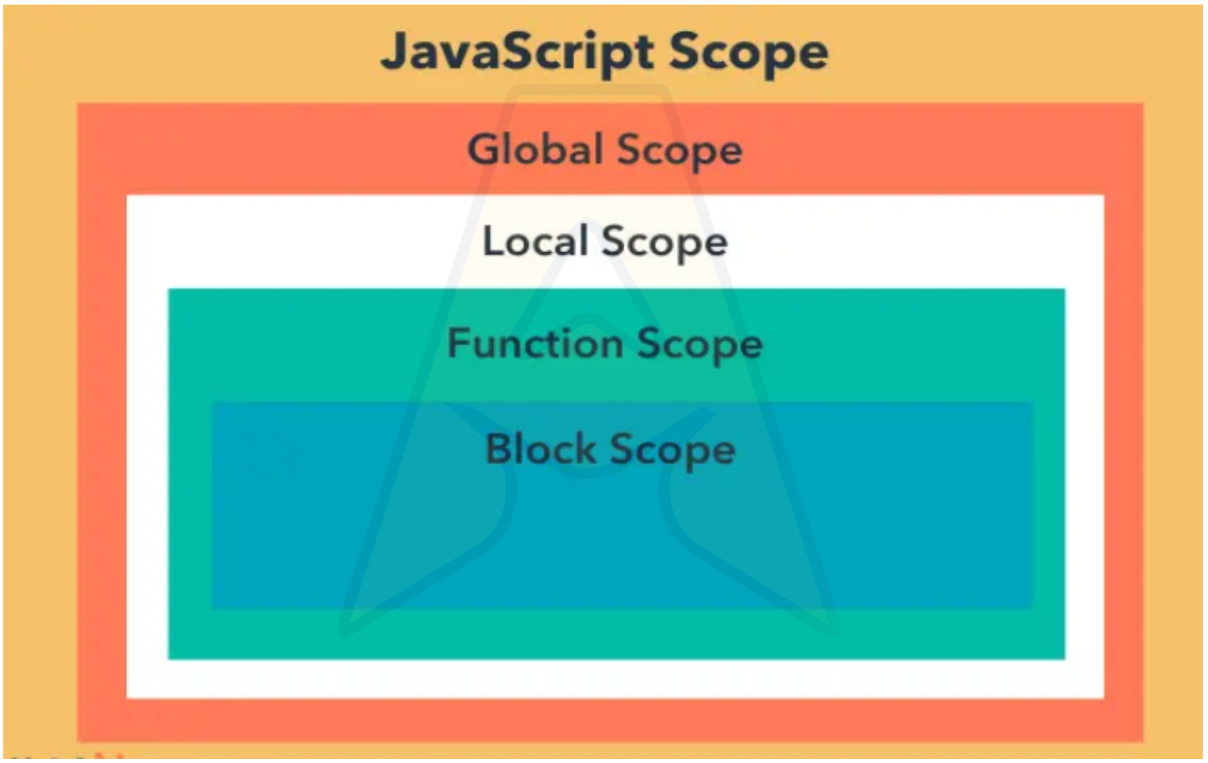
NOTES

JS Fundamentals - 1

1. Scope

Introduction to Scope in JS

The scope is the current context of execution in which values and expressions are "visible" or can be referenced. Variables and expressions that are not in the current scope cannot be used. Scopes can also be layered in a hierarchy, so that child scopes have access to parent scopes, but not vice versa.



1.2 Types of scope

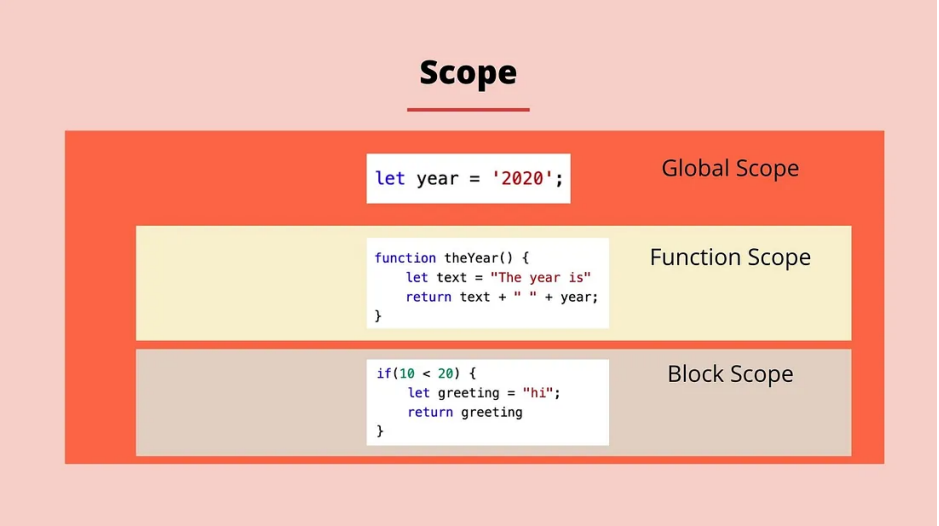
There are mainly two types of scopes:

- Global Scope
- Local Scope

Local Scope is further categorised into two sub parts:

- Functional Scope

- Block Scope



1.2.1 Global Scope

The Global Scope is used to declare variables and functions, which will be available to use throughout globally. Global variables can be accessed from anywhere in a JavaScript program.

Global scope is the entire Javascript execution environment. Any variables assigned without an explicit scope (using the var, let, or const keywords) automatically become global variables. The same is true for functions.

Example: Show the context of using global scope with JS.

Javascript:

```javascript
var globalVar = "I am a global variable";

function showGlobalVar() {
      console.log(globalVar); // Accessing the global variable inside a function.
}
// Accessing the variable "globalVar" outside the function.

// Calling the function
```

```
showGlobalVar();
```

OUTPUT:

```
PS C:\Users\Public\Aimerz\Test> node test.js
I am a global variable
I am a global variable
PS C:\Users\Public\Aimerz\Test>
```

1.2.2 Local Scope

Local scope in JavaScript refers to variables that are declared within a function. These variables are only accessible within that function and cannot be accessed outside of it.

Example: Show the context of using local scope with JS.

Javascript:

```javascript
function showLocalVar() {
    var localVar = "I am a local variable";
    console.log(localVar); // Accessing the local variable within the function
}

// Calling the function
showLocalVar();
```

OUTPUT:

```
PS C:\Users\Public\Aimerz\Test> node test.js
I am a local variable
PS C:\Users\Public\Aimerz\Test>
```

* If we try to use the declared variable outside the local scope it will throw us an error.

Benefits of Using Local Scope:

- **Encapsulation:** Keeps code within its defined boundaries, making it easier to manage and debug.

- **Avoids Conflicts:** Prevents variables from interfering with each other by confining their usage.
- **Memory Management:** Frees up memory as soon as the function execution is complete, optimizing performance.

1.2.2.1 Function Scope

Function scope in JavaScript means that variables and functions declared inside a function are only accessible within that function. This encapsulation helps avoid polluting the global scope.

Benefits of Using Function Scope :-

- **Encapsulation**: Keeps variables isolated within functions, reducing conflicts.
- **Memory Efficiency**: Frees up memory post-function execution.
- **Modularity**: Promotes reusable and maintainable code.
- **Code Clarity**: Limits scope to relevant sections, enhancing readability.

Lets now understand function scope using an example:

Example:

Code Snippet:

```javascript
function outerFunction() {
    var outerVar = "I am an outer variable";

    function innerFunction() {
        var innerVar = "I am an inner variable";
        console.log(outerVar); // Accessible
        console.log(innerVar); // Accessible
    }

    innerFunction();
    console.log(outerVar); // Accessible
    console.log(innerVar); // Uncaught ReferenceError: innerVar is not defined
}

outerFunction();
```

OUTPUT:

```
PS C:\Users\Public\Aimerz\Test> node test.js
I am an outer variable
I am an inner variable
I am an outer variable
C:\Users\Public\Aimerz\Test\test.js:12
 console.log(innerVar); // Uncaught ReferenceError: innerVar is not defined
             ^
```

- In the above output it is clearly visible that if we try to use the declaration outside the function (scope) it is not accessible in any context.

1.2.2.1 Block Scope

Block scope in JavaScript refers to variables that are accessible only within the block (denoted by curly braces {}) where they are defined. Variables declared with let and const have block scope.

Benefits of using Block Scope :-

- **Encapsulation**: Keeps variables within specific blocks, avoiding conflicts.
- **Memory Efficiency**: Frees up memory after block execution.
- **Modularity**: Enhances code organization and readability.
- **Scope Management:** Limits variable access to relevant code sections.

Code Snippet:

```javascript
{
    let blockScopedVar = "I am confined to this block";
    console.log(blockScopedVar); // Accessible here
}
// console.log(blockScopedVar); // Uncaught ReferenceError: blockScopedVar is not defined

for (let i = 0; i < 5; i++) {
    console.log(i); // Accessible here
}
```

OUTPUT:

```
PS C:\Users\Public\Aimerz\Test> node test.js
I am confined to this block
0
1
2
3
4
PS C:\Users\Public\Aimerz\Test>
```

Note: In the above example if we try to access the block scoped variable outside the scope then it will throw us an error.

JavaScript

```
{
    let blockScopedVar = "I am confined to this block";
    console.log(blockScopedVar); // Accessible here
}
//  console.log(blockScopedVar);  //  Uncaught  ReferenceError:
blockScopedVar is not defined

for (let i = 0; i < 5; i++) {
    console.log(i); // Accessible here
}

console.log(i); // Uncaught ReferenceError: i is not defined
```

OUTPUT:

```
C:\Users\Public\Aimerz\Test\test.js:11
console.log(i); // Uncaught ReferenceError: i is not defined
            ^

ReferenceError: i is not defined
    at Object.<anonymous> (C:\Users\Public\Aimerz\Test\test.js:11:13)
    at Module._compile (node:internal/modules/cjs/loader:1469:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1548:10)
    at Module.load (node:internal/modules/cjs/loader:1288:32)
    at Module._load (node:internal/modules/cjs/loader:1104:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:174:12)
    at node:internal/main/run_main_module:28:49
```

Closure

Introduction

A closure is created when a function is defined within another function and the inner function captures the environment in which it was created, specifically the variables of the outer function, even after the outer function has finished executing.

In general Javascript has 3 Scope Chains:

- It has access to its own scope (between its own curly brackets)
- It has access to the outer function scope
- It has access to the global scope

Example:

Imagine you're at a bakery, and the baker gives you a recipe (the inner function) to bake cookies. You take the recipe home. Even though you're no longer in the bakery (the outer function has finished), you still remember the ingredients and the oven temperature (variables of the outer function) the baker told you, and you can still bake the cookies.

Code:

```javascript
function outerFunction(baker) {
    let recipe = "Cookie recipe with chocolate chips"; // variable in outer scope
    return function innerFunction() {
        console.log(`${baker} gave you the recipe: ${recipe}`);
    }
}

let myClosure = outerFunction("John the Baker"); // outer function is called
myClosure();  // inner function remembers the "baker" and "recipe" variables
```

OUTPUT:

```
Node.js v20.18.0
PS C:\Users\Public\Aimerz\test> node test.js
John the Baker gave you the recipe: Cookie recipe with chocolate chips
PS C:\Users\Public\Aimerz\test>
```

Lexical Scope

Lexical scope is the concept of determining the scope of a variable based on its declaration. This means that the scope of a variable is determined by the block of code in which it is declared, not by the block of code in which it is used.

Example: Let us take a variable name fullstack and declare it in the function.

Code:

```javascript
function fullstack_function() {
        var fullatack = 'developer';
        console.log(fullstack);
}
fullstack_function();
```

OUTPUT:

```
Node.js v20.18.0
PS C:\Users\Public\Aimerz\test> node test.js
developer
```

*The variable declared inside the function will only be accessible within its lexical scope and not beyond it.

Lets try to access the variable fullstack outside its lexical scope.

Example:

```javascript
function fullstack_function() {

      var fullstack = 'developer';
      console.log(fullstack);
  }

  fullstack_function();
  console.log(fullstack);
```

OUTPUT:

```
Node.js v20.18.0
PS C:\Users\Public\Aimerz\test> node test.js
developer
C:\Users\Public\Aimerz\test\test.js:8
  console.log(fullstack);
              ^

ReferenceError: fullstack is not defined
    at Object.<anonymous> (C:\Users\Public\Aimerz\test\test.js:8:15)
    at Module._compile (node:internal/modules/cjs/loader:1469:14)
```

In the above output, we can clearly see the observation of lexical scope. The first output is from execution of fullstack_function() function and the error is reflecting because we are trying to access the variable fullstack outside its lexical scope.

Scope Chaining

Scope chaining is a hierarchical structure in JavaScript that defines the order in which JavaScript looks for variables.

Let us understand the concept of scope chaining with an example.

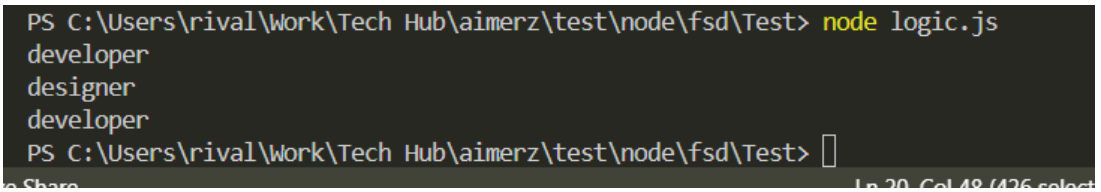
Example:

JavaScript

```javascript
var fullstack = 'developer';
function fullstack_function() {
    console.log(fullstack); // Accesses 'developer'
}

function outerFunction() {
    var fullstack = 'designer';

    function innerFunction() {
        console.log(fullstack); // Accesses 'designer'
    }

    innerFunction();
}

fullstack_function();
```

```
outerFunction();
console.log(fullstack); // Accesses 'developer'
```

OUTPUT:



Explanation:

- **console.log(fullstack);** inside innerFunction accesses the fullstack variable defined in the outerFunction scope and outputs designer.
- **console.log(fullstack);** inside fullstack_function accesses the global fullstack variable and outputs developer.
- **console.log(fullstack);** outside of any function accesses the global fullstack variable and outputs developer.

Abstraction & Encapsulation

Closures in JavaScript allow functions to have access to variables from an outer function even after the outer function has closed. This concept helps in encapsulating data and creating abstractions.

Let us understand the above condition with an example:

JavaScript

```
var fullstack = 'developer'; // Global scope

function fullstack_function() {
    console.log(fullstack); // Accesses 'developer'
}

function outerFunction() {
```

```
    var fullstack = 'designer'; // Local scope for outerFunction

    function innerFunction() {
        console.log(fullstack); // Accesses 'designer'
    }

    return innerFunction;
}

fullstack_function(); // Outputs: developer

var closureFunction = outerFunction();
closureFunction(); // Outputs: designer

console.log(fullstack);  // Accesses 'developer'
```

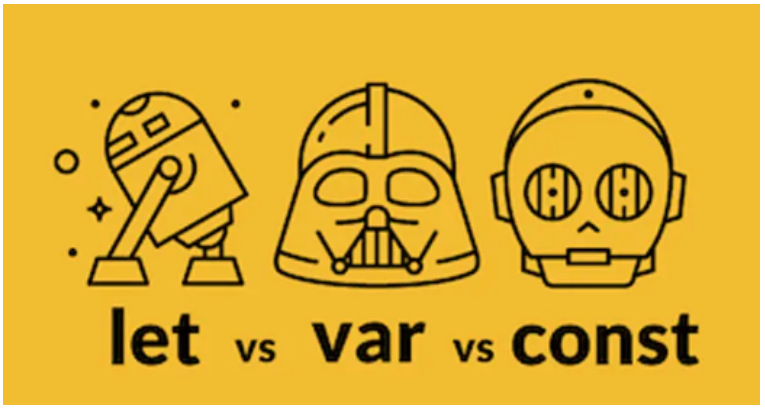
OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
developer
designer
developer
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

let var & const

Introduction to let and const

let and **const** are two ways to declare variables in JavaScript that provide block scope. They were introduced in ECMAScript 6 (ES6) to offer more robust scoping compared to the traditional var.



Key Properties of Let:

- Variables declared with let are limited to the block (e.g., inside {}) in which they are defined.
- We can reassign all the let variables.

Key Properties of const:

- Block Scope: Like let, const variables are also block-scoped.
- No Reassignment: You cannot reassign const variables once they are defined.

Need of let and const

let and const were introduced in ES6 (ECMAScript 2015) to address limitations and issues with var.

Key reasons why let and const are often preferred over var:
Block Scope:

- **Problem with var:** var is function-scoped, meaning if declared in a function, it's accessible throughout the entire function. If declared in a block (e.g., within an if statement or loop), it's still accessible outside of that block, which can lead to unexpected behavior.

Solution with let and const: Both let and const are block-scoped, which means variables declared within a block { ... } are only accessible

within that block. This scoping helps prevent accidental modifications or accessing variables in unintended places.

Example:

```javascript
if (true) {
    var x = 10;
    let y = 20;
}
console.log(x); // 10 (accessible outside the block)
```

Out

Difference between let and const vs var

S no.		Scope	Hoisting	Reassignable
1.	let	Accessible only within the block { ... }	let is hoisted but not initialized	We can reassign let variables but can't redeclare them in the same scope.
2.	var	Accessible within the function where it was declared.	`var` declarations are hoisted.	We can reassign and redeclare var variables within the same scope.
3	const	Block-scoped, like let	const is also hoisted but not initialized.	const variables cannot be reassigned or redeclared within the same scope.

THANK YOU

AIMERZ.ai
Aim.Act.Achieve

Stay updated with us!

Vishwa Mohan

Vishwa Mohan

vishwa.mohan.singh

Vishwa Mohan