Scope

A scope in JavaScript defines what variables you have access to. There are two kinds – global and local scope.

#### #Global scope

If a variable is declared outside all functions or curly braces ({}), it is said to be defined in the **global scope**.

This is true only with JavaScript in web browsers. You declare global variables in Node.js differently

**Collisions:**Although you can declare variables in the global scope, it is advised not to. This is because there is a chance of naming collisions, where two or more variables are named the same. If you declared your variables with constor let, you would receive an error whenever a name collision happens. This is undesirable.

If you declare your variables with var, your second variable overwrites the first one after it is declared. This also undesirable as you make your code hard to debug.

#### #Local Scope

Variables that are usable only in a specific part of your code are considered to be in a local scope. These variables are also called **local variables**.

In JavaScript, there are two kinds of local scope: function scope and block scope.

#### #Function scope

When you declare a variable in a function, you can access this variable only within the function. You can't get this variable once you get out of it.

#### #Block scope

When you declare a variable with const or let within a curly brace ({}), you can access this variable only within that curly brace.The block scope is a subset of a function scope since functions need to be declared with curly braces (unless you're using [arrow functions](https://zellwk.com/blog/es6/#arrow-functions) with an implicit return).

**#HOISTING**

* ‘var’ and ‘function’ can be accessed in limited capacity before they are declared
* Execution context is created in 2 phases
  + Creation phase: memory space is setup for variable and functions
  + Execution phase: code is read and executed line by line

Function DECLARATION: will be hoisted

Functions, when declared with a **function declaration**, are always hoisted to the top of the current scope.

console.log(calcRectArea(5, 6)); //30

function calcRectArea(width, height) {

return width \* height;

}

Function EXPRESSION: will NOT be hoisted

When declared with a **function expression**, functions are NOT hoisted to the top of the current scope.

console.log(getRectArea(3,4)); // Uncaught TypeError: getRectArea is not a function

var getRectArea = function(width, height) {

return width \* height;

}

Hoisting VARIABLES:

**var/function** are initialised with ‘undefined’(placeholder)

console.log(aaa); //undefined (memory was allocated, engine knows it exists, it doesn’t know it’s value until ‘second pass’)

var aaa = "Hello World!!";

**let/const/class** stay unitialised. This means that a ReferenceError exception is thrown when you try to access it this way. let and const hoist but you cannot access them before the actual declaration is evaluated at runtime.

console.log(aaa; // Uncaught ReferenceError: aaa is not defined

let aaa = "Hello World!!";

#### #Nested scopes

When a function is defined in another function, the inner function has access to the outer function's variables. This behavior is called **lexical scoping**.However, the outer function does not have access to the inner function's variables.

To visualize how scopes work, you can imagine one-way glass. You can see the outside, but people from the outside cannot see you. If you have scopes within scopes, visualize multiple layers of one-way glass.

### Closures

Whenever you create a function within another function, you have created a closure. The inner function is the closure. This closure is usually returned so you can use the outer function's variables at a later time.

functionouterFunction(){

const outer =`I see the outer variable!`

functioninnerFunction(){

console.log(outer)

}

returninnerFunction

}

outerFunction()()

Since closures have access to the variables in the outer function, they are usually used for two things:

1. **To control side effects**
2. **To create private variables**

#### #Controlling side effects with closures

Side effects happen when you do something in aside from returning a value from a function. Many things can be side effects, like an Ajax request, a timeout or even a console.log statement:

function(x){

console.log('A console.log is a side effect!')

}

When you use closures to control side effects, you're usually concerned with ones that can mess up your code flow like Ajax or timeouts.

Let's say you want to make a cake for your friend's birthday. This cake would take a second to make, so you wrote a function that logs made a cake after one second.

functionmakeCake(){

setTimeout(\_ => console.log(`Made a cake`),1000)

}

As you can see, this cake making function has a side effect: a timeout.

Let's further say you want your friend to choose a flavor for the cake. To do so, you can write add a flavor to your makeCake function.

functionmakeCake(flavor){

setTimeout(\_ => console.log(`Made a ${flavor} cake!`),1000);

}

When you run the function, notice the cake gets made immediately after one second.

makeCake('banana')// Made a banana cake!

The problem here is that you don't want to make the cake immediately after knowing the flavor. You want to make it later when the time is right.

To solve this problem, you can write a prepareCake function that stores your flavor. Then, return the makeCake closure within prepareCake.

From this point on, you can call the returned function whenever you want to, and the cake will be made within a second.

functionprepareCake (flavor) {

return function () {

setTimeout(\_ => console.log(`Made a ${flavor} cake!`),1000);

}

}

constmakeCakeLater = prepareCake('banana')

// And later in your code...

makeCakeLater() // Made a banana cake!

That's how closures are used to reduce side effects – you create a function that activates the inner closure at your whim.

#### #Private variables with closures

As you know by now, variables created in a function cannot be accessed outside the function. Since they can't be accessed, they are also called private variables.

However, sometimes you need to access such a private variable. You can do so with the help of closures.

function secret(secretCode) {

return {

saySecretCode () {

console.log(secretCode);

}

}

}

consttheSecret = secret('CSS Tricks is amazing');

//we save the secret message in variable and pass it to secret() function

theSecret.saySecretCode();

//to access the secret message we call the variable we stored it in and prepend name of the inner function saySecretCode // 'CSS Tricks is amazing'

saySecretCode in this example above is the only function (a closure) that exposes the secretCode outside the original secret function. As such, it is also called a **privileged function**.

//ex2

functiontajemstvi(tajnyKod){

return {

dejMiTajnyKod(){

console.log(tajnyKod);

}

}

}

constulozeneTajemstvi = tajemstvi('Je tajnezezabymajidveoci!');

ulozeneTajemstvi.dejMiTajnyKod(); //Je tajnezezabymajidveoci!

**Summary:**

Scopes and closures aren't incredibly hard to understand. They're pretty simple once you know how to see them through a one-way glass.

When you declare a variable in a function, you can only access it in the function. These variables are said to be scoped to the function.

If you define any inner function within another function, this inner function is called a closure. It retains access to the variables created in the outer function.