4-11. Scope

A scope is the part of a program in which variables and functions are **visible and accessible**. *4-11-1*

We have seen a little about scopes when talking about variables with `var` and `let`. *4-11-2*

So, when you declare a variable or a function, it will be either in **global** or **local** scope. *4-11-3*

global scope *4-11-4*

Variables or functions in the global scope can be **accessed from everywhere** within the program.

Both on same level or inside other functions.

An undeclared variable will automatically be in **global scope**, no matter where it is defined. *4-11-5*

```
function myFunction() {
    someValue = 3;
    console.log(someValue); // Works fine
}

myFunction();

console.log(someValue); // Works fine
```

Undeclared variable: Variable created without `var` or `let` keyword.

We can access the variable from outside the function.

A variable declared outside of any function has **global scope**. *4-11-6*

```
//code starts here
let someValue = 3;

console.log(someValue); // Works fine

function myFunction(){
    console.log(someValue); // Works fine
}
```

It is visible from its point of declaration to the end of the program.

So is a function created in the root, outside of any other function. *4-11-7*

```
//code starts here
function myGlobalFunction() {
    console.log("Can be accessed anywhere");
    function myNestedFunction() {
        console.log("Can't be accessed from outside myGlobalFunction");
    }

    myNestedFunction(); // Works fine
}

myGlobalFunction(); // Works fine

myNestedFunction();
// Uncaught ReferenceError: myNestedFunction is not defined
```

In the case of a web browser, everything in the **global scope** is attached to the **window** object. *4-11-8*

The global scope lives as long as your program is running, it is accessible during this time. *4-11-9*

Local scope *4-11-10*

A **local scope** is a smaller part of the program in which variables and functions lives.

A function is for example a smaller, **local scope** for a variable in JavaScript.

Variables declared in a local scope can only be accessed within this scope. *4-11-11*

```
function myFunction(){
    let someValue = 3;
    console.log(someValue); // Works fine
}

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

But, we can still use things that are in the **outer scope** inside the function. *4-11-12*

Modification of a variable for example.

Meaning the below will work just fine *4-11-13*

```
var x = 3;

function myLocal() {
    x = 76;
}

myLocal();

console.log(x); // 76
```

But this will not *4-11-14*

```
function myLocal() {
    var x = 4;
}

myLocal();

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

Local scope lives as long as a function is called or block is executed. *4-11-15*

Recap var vs let *4-11-16*

The preferred way to declare variables is using the `let` keyword.

We usually want to achieve as **small scopes** as possible in our code.

Variables declared with the `var` keywords are only local in function scope. *4-11-17*

```
var x = 3; // global

function y() {
    var u = 5; // local, function scope
}

for(var i = 0; i < 5; i++) {
    // i is global and can be accessed from outside the loop
    console.log(i); // Works fine
}

console.log(i); // Works fine
```

So, declaring a variable using `var` it becomes **global** unless it is declared within a function. *4-11-18*

`let` lets us work with smaller scopes. *4-11-19*

```
let x = 3; // global

function y() {
    let u = 5; // local, function scope
}

for(let i = 0; i < 5; i++) {
    // i is local within the scope of the loop
    // and cannot be accessed from outside
    console.log(i); // Works fine
}

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

When `let` is used inside a block, accessibility of the variable is limited to that block. *4-11-20*

```
let x = 3; // global

if(x === 3) {
    let y = 6; // The scope is inside the if-block
}

console.log(y);
```

*Creating a variable with `var` inside an if-block would result in it having **global scope**.*

Q Why scopes, why not have everything available everywhere in your code? *4-11-21*

A There's a principle called **The principle of *Least Access***. *4-11-22*

When running a program it is a good rule that users only should have access to what they need for the task at hand. *4-11-23*

This facilitates troubleshooting for example. *4-11-24*

If an error occurs it would be hard to identify where the error happened and what caused it, if everything had access to everything.

If we limit the access we can more easily notice what the error is and where to look for it. *4-11-25*

Having scopes also facilitate naming problems when we have variables with the same name but for different purposes. *4-11-26*

4-12. Closure

A **closure** is an inner function that has access to the outer (enclosing) function's **variables** and **scope chain**. *4-12-1*

The closure has three scope chains. It has access to: *4-12-2*

- **its own scope (variables defined within its curly brackets)**
- **the outer function's variables**
- **and the global variables.**

You create a closure by **adding a function inside another function**. *4-12-3*

4-12-4

```
function display(firstName, lastName) {
    var title = " is the Master of the Universe!";
    function addTitle() {
        return firstName + " " + lastName + title;
    }

    return addTitle();
}

display("Therése", "Barmer");
```

Since the local variable `title` is used by the inner function `addTitle`, it will stay in scope.

An example of closure are **callback functions**. **jQuery** makes frequent use of callback functions: *4-12-5*

```
$("button#clickme").click(function() {
    console.log("Yep, I was clicked");
});
```

In the example above, we are passing an anonymous function as a parameter to the `click` method.

The anonymous function that we passed in will be called by the `click` method. Note that we are only passing the *function definition*, not executing the function. *4-12-6*

So, in plain words: A **callback** is a function called at the **completion of a given task**. In our case, when the button is clicked, "Yep, I was clicked" will be logged out to the console. *4-12-7*

This works because functions are *first class citizens* in JavaScript. This means that functions: *4-12-8*

- **can be passed as arguments to other functions**
- **returned as values from other functions**
- **assigned to variables**
- **and stored in objects.**