# Part 3

## Scope

In JavaScript there are two types of scope:

* Local scope
* Global scope

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

Scope determines the accessibility of variables.

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

### Local Scope

* Variables declared within a JavaScript function, become **LOCAL** to the function.
* Local variables have **Function scope**: They can only be accessed from within the function.
* 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.

// code here CANNOT access carName

function myFunction() {

let carName = 'Volvo';

// code here CAN access carName

}

### Global Scope

* A variable declared outside a function, becomes **GLOBAL**.
* A global variable has **global scope**: All scripts and functions on a web page can access it.

let carName = 'Volvo';

// code here can access carName

function myFunction() {

// code here can also access carName

}

## Closure

* An inner function that has access to the outer functions scope chain
* Closure has 3 scope chains
  + Its own scope
  + Outer function scope
  + Global scope
* Allows us to write code that is:
  + Creative
  + Expressive
  + Concise

function showName(firstName, lastName) {

const nameIntro = 'Your name is ';

function makeFullName() {

return nameIntro + firstName + ' ' + lastName;

}

return makeFullName();

}

showName('Adam', 'West');

Closure with a function inside another function.

One of the most important features is that the inner function still has access to the outer functions variables even after the outer function has returned.   
When functions execute, they use the same scope that was in effect when they were created, this means that even after the outer function has returned the inner function still has access to the variables later in your program.

function celebrityName(firstName) {

var nameIntro = 'This celebrity is ';

function lastName(theLastName) {

return nameIntro + firstName + ' ' + theLastName;

}

return lastName;

}

var awName = celebrityName('Adam');

awName('West');

Closures have access to the outer functions variable even after the outer function returns

They do not store the actual value, closures get more interesting when the value of the outer functions’ variable changes before the closure is called. This powerful feature can be harnessed in creative ways, such as this private variable example.

function celebrityID() {

let celebrityID = 999;

return {

getID: function () {

return celebrityID;

},

setID: function (theNewID) {

celebrityID = theNewID;

}

}

}

const awID = celebrityID();

awID.getID();

Closures store references to the outer function’s variables

## Hoisting

* In JavaScript, a variable can be declared after it has been used
* In other words; a variable can be used before it has been declared
* Hoisting is JavaScript's default behaviour of moving all declarations to the top of the current scope (to the top of the current script or the current function)

These two blocks of example code produce the same result:

var x;

x = 5;

console.log(x);

x = 5;

console.log(x);

var x;

### Hoisting with Functions

function myFunc(){

console.log("Hi");

}

Declaration

const expression = function myFunc() {

console.log("Hi");

}

Expression

Hoisting works with methods too, in the expression example if we try to call the method before it’s initialized, we’ll get undefined, in the first example it doesn’t matter when we call it.

In the expression example we can also re-assign what expression is, in case we don’t need the method anymore or want it to be another method.

In the second example the function could call itself without anything else being able to call it directly, which is handy – its better.