All the content below comes from Kyle Simpson and his fantastic serie of books you don't know JS.

Javascript is considered a **dynamic** or interpreted language in comparison to compiled language like C# which means there is a compilation phase beforehand which allows usually your IDE to tell you what's wrong with your code beforehand

That being said Javascript performs also some of the same steps that the compiled language but just before execution time. There are 3 main phases going on :

1. **Tokenizing/Lexing** : breaking up a string of character into meaningful chunk
2. **Parsing** : taking an array (stream) of tokens and turning into a tree of nested element (tree called AST - abstract syntaxt tree)
3. **Code-Generation** : process of taking an AST and turn it into executable code

There are 3 main actors which handle those operations:

- **Engine** : responsible for start to finish compilation and execution of our code
- **Compiler** : handles work of parsing and code generation
- **Scope** : collects and maintains a lookup list of all the declared variable

Let's dive in for some scenarios what does it mean :

```
var a = 2
//which is equivalent to
var a
a = 2

// 1. first compiler declares the variable
// if not previously declared in current scope

// 2. Engine takes the compiler code and looks up
// for the variable in scope to assign it the 2 value if found
```

There are 2 kind of lookups : RHS and LHS (right hand side and left hand side of the assignment operator)

- RHS : we're looking for the value of the variable
- LHS : we're looking for the variable container itself - usually to assign it a value

If variables are not found in current scope, Engine asks to outside scope

Some more scenarios to practice :

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

//RHS lookup cannot find it in the scope
```

```
'use strict'
b = 2
console.log(b) // Uncaught ReferenceError: b is not defined

//LHS fails as no b declaration
//RHS lookup cannot find it in the scope
```

Without strict mode, global scope will create the variable for you

```
b = 2
console.log(b) // 2

//LHS on b creates a global variable when not found
//RHS can then get the value and pas it along to console.log()
```

```
var c = 3
c() //Uncaught TypeError: c is not a function

// LHS and RHS succeed
// trying to do something impossible with the value found => TypeError
```

Lexical Scope

Lexical scope means that scope is only defined by **author time** decisions of **where functions are declared**:

```
function outerScope(a){
 console.log(a)
    function innerScope(){
        var a = 3
        console.log(a)
    }
}
outerScope(2) // 2 3
```

Scope Lookup stops since it finds the **first match**. (always begin from the innermost to the outermost identifier) The same name can be specified at multiple layers of nested scope.

If 2 same identifiers are present, the innermost one will 'shadow' the outermost one. It's good to know that there are 2 mechanisms that can cheat the lexical scope : eval(...) and with.

Therefore scope consists in a serie of bubbles. There are several ways to create new bubbles/scope.

- Function scope
- Block Scope - let and const

Function scope :

A function creates a specific scope inside of which all variables declaration are contained to the scope. All variables declared this way can be accessed within the function and within nested function scope. There are several advantages to this functional scope

- Principe of least privilege : recommendation in design software to expose only what is minimally necessary.
- Collision avoidance : avoid collision between 2 different identifiers with the same name

```
function outerScope() {
    var a = 10
    console.log(a)
    function innerScope() {
        var a = 3
        console.log(a)
    }
    innerScope()
}
innerScope() // Uncaught ReferenceError: innerScope is not defined
outerScope() // 10 3
```

A common example of similar design are some libraries, which depends on a specific namespace (usually an object) like lodash (_.) or jQuery ($.). Those namespace are used to avoid collision between different properties / methods Nowadays libraries use module pattern and dependency manager avoiding the need to create global name.

Functional scoping is powerful but as a few drawbacks, as you need to create a function (named or anonymous) and this pollutes the global scope. JS provides a solution for this

```
(function outerScope() {
    //...code
})()
```

Having **()** in front of the function keyword, change the function **nature** :

- **function declaration** : if function keyword is the first word in statement
- **function expression** : if function is preceded by other keyword ((function... or const add = function()... or setTimeout(function()...)

There are a few differences between the 2 :

- **function declaration are hoisted**, not function expression
- **function expression does not create an identifier in the parent scope**, it hides it inside its own scope The example mentioned above immediately invoking the function is called **IIFE** (immediately invoked function expression)

Block as scope - let and const

Let and const introduced by ES6 provides block scope.

Those keywords attach variable declaration to the scope of whatever block they are contained in.

```
if(true){
    var a = 1
    let b = 2
    const c = 3
    console.log(a,b,c) // 1 2 3
}
console.log(a,b,c) // Uncaught ReferenceError: b is not defined (same for c)
```

Hoisting

Hoisting is the process through which JS engine 'move' variable and function declarations to the top of the code.

```
a = 2
var a
console.log(a) // 2
```

Some clarifications :

- hoisting is **per scope** : it moves the variable declaration to the top of the enclosing scope
- variable declaration with **let/const are not hoisted**
- function expression are not hoisted neither
- function are hoisted first and then variable declarations

Closure

Closure is when function is able to remember and access its lexical scope even when that function is executing outside of its lexical scope. Or simpler as W3school puts it : A closure is a function having access to the parent scope, even after the parent function has closed.

```
function addTwo(number){
    var a = number + 2
    function multiplyByTwo(){
        console.log(a * 2)
    }
    return multiplyByTwo
}
const six = addTwo(1)
six() // 6

// After addTwo() is executed, we might have expected its scope
// to be garbage collected as not 'in use ' anymore.
// Even if multiplyByTwo is called outside of its declared scope (through six()),
// it still has access to the inner scope of addTwo. That's what we call closure.
```

Real life examples : Callbacks and Module Pattern

```
// CALLBACKS
function wait (message) {
    setTimeout(function timer() {
        console.log(message)
    }, 1000)
}
wait('Thanks Kyle')
//Timer has a scope closure over the inner scope of wait,
//which maintains a reference to the message variable
```

```
//MODULE PATTERN
function module() {
    const name = "js journey"

    function getName() {
        console.log(name)
    }

    return {
        getName:getName
    }
}
const firstModule = module()
firstModule.getName()
```

Calling module() assigns its return value (the object with getName as a method) to firsModule. From there we can then access getNames. The advantage of this pattern is that it only exposes as public API what you've chosen to return as a value

Module pattern requires :

- An outer wrapping function being invoked to create the enclosing scope
- The return value of the wrapping function must include a reference to at least one inner function