# Hoisting & Closure

## Hoisting:

Hoisting is a term you will not find used in any normative specification prose prior to ECMAScript® 2015 Language Specification. Hoisting was thought up as a general way of thinking about how execution contexts (specifically the creation and execution phases) work in JavaScript. However, the concept can be a little confusing at first.

Conceptually, for example, a strict definition of hoisting suggests that variable and function declarations are physically moved to the top of your code, but this is not in fact what happens. Instead, the variable and function declarations are put into memory during the *compile* phase, but stay exactly where you typed them in your code.

Ex:01

| | |
|---|---|
| **`x()`**<br>**`def ():`**<br>**`    return 123`**<br>**`        will show an error`** | **`def ():`**<br>**`    return 123`**<br>**`x()`**<br>**`        will run the code`** |

Python being an interpreted language and it is a one parse system so it would execute one command at a time. So, when it reads the first line x(), it will show an errors stating that x is not defined. But if def(): return 123 is come first and then x(), it will not show error.

We are defining the function at line number 3 and calling the function at line number 1. Python is also an interpreted language it will go line by line; it also has one pass system, meaning goes through the code the time of executing. However, JavaScript is a two-pass system.

In Python at line number 1 immediately at well throw an error that X is not defined. But when we put the function above and then call function then next function will be executed. it is going through this happens because Python is one party system do you code and at the same time it executes the code but in the other case it will say that you have called the function and it would call the function and the moment it calls to function it throws a typo error. this is because the moment it goes through one line is starts executing.

But, in JavaScript, being a two-pass system, it will first go through the entire code, trying to understand everything declare in the global context. Then in the second pass, JavaScript will actually start executing the code.

```
var avc

function name(params) {
    var asd
}
```

**Now,**

```
var avc

function x(params) {

}

function y(params) {

}
```

We can see that function x and y are top level elements. The var 'avc' gets declared in the window object. When we say top level declaration, 'avc' and functions x and y get attached to the window object. Even if we put **`var avc`** at the bottom, and we add console.log (avc) at the top, in Python it would have thrown an error. But in JavaScript being two-pass system, in the first pass, it will see what are the global declaration. 'avc' will be one variable declared and functions x & y. So, Javascript in the memory, will take **`var avc`** right at the top and will push the console.log at the bottom of the page.

| The code that we have written | The code that JavaScript understands |
|---|---|
| `console.log (avc)` | `var avc` |
| `function x(params) {` | `function x(params) {` |
| | |
| `}` | `}` |
| `function y(params) {` | `function y(params) {` |
| | |
| `}` | `}` |
| `var avc = 10` | `console.log (avc)` |
| | `avc = 10` |

If we execute the code, we would get an output in console as undefined. The solution came in ES6 (const & let) that solves this hoisting issues.

`var myVariable;` → just declaration and no value is given to it.

`var myVariable = 10` → variable has been declared and has been initialized.

**INPUT:**

```
print()
show()

console.log(myvariable)
function print() {
     console.log("printed")
}

console.log(abc)
function show(params) {
     console.log(myvariable + abc)
}

var myVariable = 100
var abc = 20
```

**OUTPUT:**

```
Printed

Nan

Undefined

Undefined
```

**Understanding:**

The code that we have written

```
print()
show()

console.log(myvariable)

function print() {
     console.log("printed")
}

console.log(abc)

function show(params) {
     console.log(myvariable + abc)
}
```

The code that JavaScript understands after the first pass

```
var myVariable
var abc

function print() {
     console.log('printed')
}
function show() {
     console.log(myVariable + abc)
}

print()
show()
```

```
var myVariable = 100
var abc = 20
```

```
console.log(myVariable)
console.log(abc)

myVariable = 100
abc = 20
```

In the second pass, JavaScript will start executing the code.

Both variables, myVariable & abc will be assigned primitive value undefined. Then it will read the print function and show function. Now, at line 10 print() function declaration came first, so the code knows what is the logic that needs to be executed when the function is called. Then the control will go back to the show() function called below the print() function and will return back to line 7 and the show() function logic is executed.

Since it is executed it will go to console.log(myVariable). As the variable value is not assigned to it yet, it will show undefined in console and the same goes for console.log(abc).

After this the variables are assigned their values. Now if we enter the variables directly in the console, it will show the values assigned to them. Now that the values have been assigned to the variables, if we use console.log(myVariable + abc); then the output would show the sum of 100 & 20.

We understand that we can pass a function as a parameter to another function as an argument. Similarly, like we are passing function, we can return function from another function.

```
function abc(callback) {
     callback()
}
function xyz() {

}

abc(xyz)
```

When abc is called, this callback variable will be called back as a function.

```
function addCreater() {

     function add(a,b) {
          console.log(a + b)
     }
     return add
}
```

We have declared a function inside another function. Now that my add() function is declared inside addCreator() function, so we can return add.

If we add **`var plus = addCreator()`**

```
function addCreater() {

     function add(a,b) {
          console.log(a + b)
     }
     return add
}
var plus = addCreator()
console.log(plus)
```

If we have return add, then it would return the function definition, but if we have add() then it will only return the value.

If we give plus(10, 20); then the entire definition is given the a and b values. If we are passing a function as a parameter and that function is return parameter, then that function is called an higher order function.

Let us add another function called subtract () which is passed to addCreator.

```
function addCreator(callback) {
     function add(a,b) {
          console.log(a + b)
     }
     return add
}

function subtract() {
     console.log('Subtract')
}

var plus = addCreator(subtract)
```

```
console.log(plus)
plus(10, 20)
```

```
Output:

Subtract

    f add(a,b) {
        console.log(a + b)
    }
```

We can use anc as a callback function. Using the parameter given in addCreator function, if we call the same parameter as a function then it is called a callback function.

## Closures:

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.