

- Call Stack maintains the order of execution of execution contexts.
- It handles everything to manage this execution context creation, deletion, and the control.
- Other names of Call Stack
  1. Execution Context Stack
  2. Program Stack
  3. Control "
  4. Runtime "
  5. Machine "

## ✱ Episode-3

### ✱ Hoisting in Javascript

- Hoisting is a Javascript mechanism where variables and function declarations are moved to the top of their scope before code execution.
- As we learnt in Execution Context (2 phases),
  - Before the code is executed, the variables get initialized to undefined.



- Arrow functions act as variables and get "undefined" during the memory creation phase while functions actually get run (whole func. code is stored)

→ 2 golden rules:

- 1) Variable declarations are scanned and are made undefined.
- 2) Function declarations are scanned and are made available.

→ e.g. Code:

- `var x = 7`

```
function getName() {  
  console.log("Namaste JS")  
}
```

```
getName()  
console.log(x)
```

⇒ Output:

```
Namaste JS  
7
```

First, the func. gets called and then `x` is console'd (printed) ⇒ Simple



- `getName()`  
`console.log(x)`

`var x = 7`

```
function getName() {
  console.log("Namaste JS")
}
```

⇒ Output :

Namaste JS  
undefined

As mentioned earlier, the variables and functions are allocated undefined and whole func. code respectively in the memory.

So, before execution, `x` is undefined and `getName()` is called and executed.

- If `var x = 7` is removed and we run the code, we will get error: `x` is not defined.

- `getName()`  
`console.log(getName)`

⇒ Namaste JS

```
f getName() {
  console.log("Namaste JS")
}
```

↗ whole func. is printed

- Arrow functions:

```
var getName = () => {  
  console.log("Namaste JS")  
}
```

⇒ Now, if we do `getName()` before this, it will console "undefined".

- ```
var getName = function() {  
  console.log("Namaste JS")  
}
```

⇒ It will also be stored as "undefined" during memory creation phase