- EXECUTION CONTEXT
- HOISTING
- VARIABLE HOISTING
- FUNCTION HOISTING

Javascript Execution Context:

PART - 1

* Concepts like "Hoisting", "this" is a mystery to many javascript developers, but to unravel & to demystify / clarify the concept, the basic concept is execution context

→ Execution context is the environment that enables Javascript code to execute. It decides which piece of code gets what level of access.

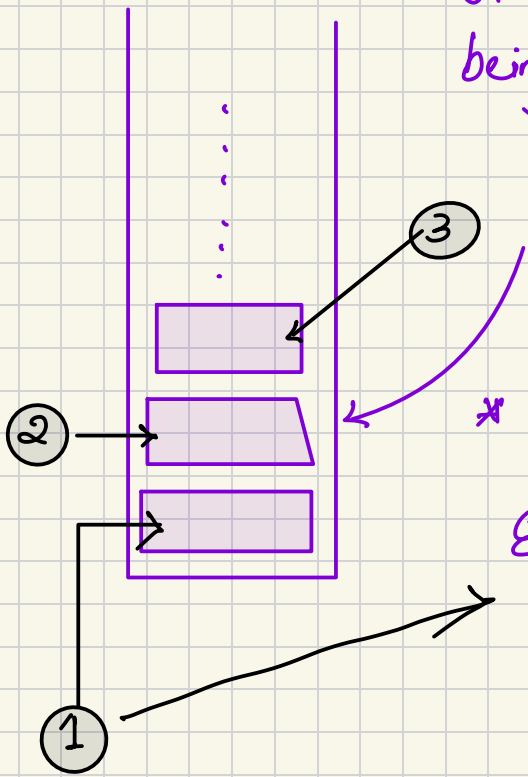
→ Variables & objects get Stored in the Memory Heap.

[We will learn more about call Stack & Memory]

the Stored data will then be used during the code execution.

Call Stack / Execution Stack:

* It has the functions that are being called & executed by the JS Engine.



* LIFO Mechanism

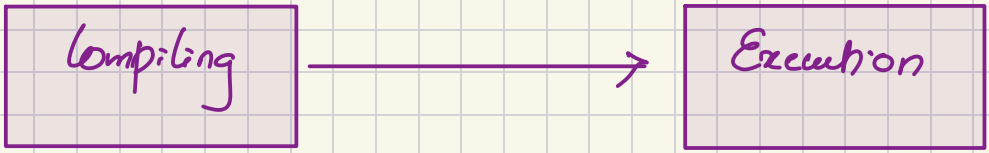
Ex: ① fn main() {
    hello();
}

② fn hello() {
    print("Hello");
}

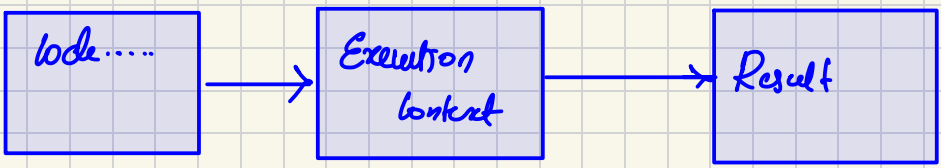
③ fn print(greeting) {
    console.log(greeting);
}

- Javascript is Single threaded
- Javascript Engine has only one Call Stack
- Javascript can only do one thing at a time.

Javascript runs in two simple steps:



When the javascript code runs, an execution context is created. When the execution context is ready, execution step starts.

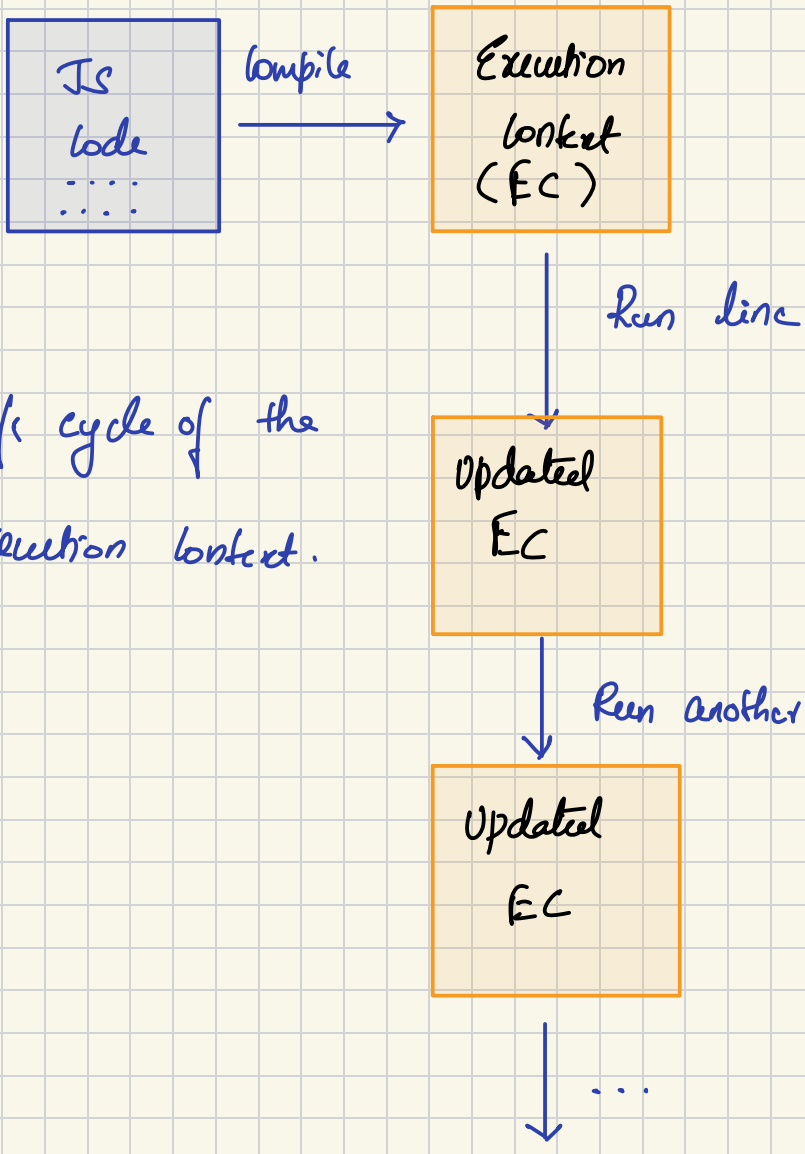


* The execution context has many things, lets focus on the main things for the sake of understanding

Execution context

* Variable environment
* Lexical environment
* outer
* this

* The execution context updates with each line since each line can be a potential changer of the context.



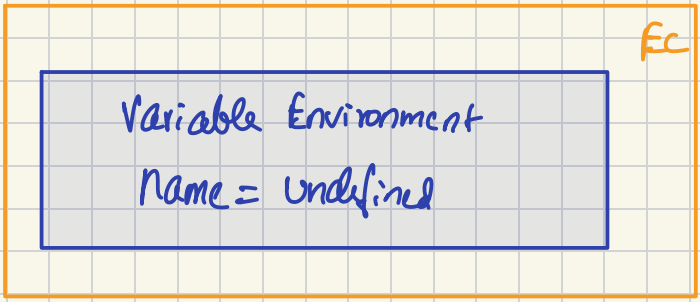
Life cycle of the execution context.

Lets understand with an example

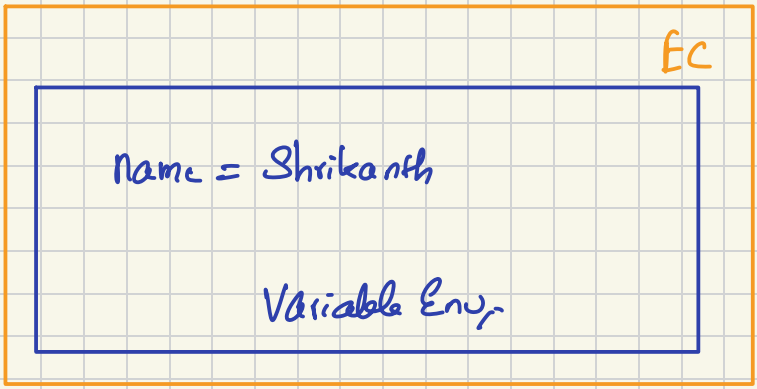
```
L1 → var name = "Shrikanth";
L2 → console.log(name)
```

Phase 1: Compiling

Variable name is created & initialized



Phase 2: Execution Step -



* The value is assigned to the variable & the variable env; is also updated simultaneously.

Code moves to the next line.

```
console.log(name)
```

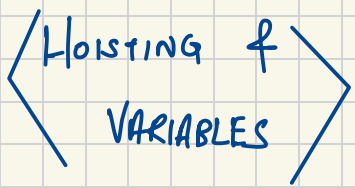
* Console starts looking for the name variable in the variable env; & indeed it finds it & prints the value.

There is no more code to execute, the EC is removed.

HOISTING & EXECUTION CONTEXT!

Consider the below code:

```
Line 1  console.log(name);
Line 2  var name = "shrikanth";
```



This will log undefined, but why?

During compilation phase, line 1 is skipped since, it is not related to variable declaration. Variable name is created & compilation phase ends.

During execution phase, the console now looks for the variable name, & it is at-this-point holding the value of "undefined" & hence it logs undefined.

Now, the flow moves to line 2 & it updates value of name. The process ends & the EC is popped off.

we call this process as hoisting because it feels as though the variable name is moved to the top. But under the hood there is no actual "hoisting", instead, its the two phases.

Due to this, the code would feel the code to be:

```
var name = undefined;
console.log(name);
name = "Shrikanth";
```

HOISTING & FUNCTIONS

Functions can be declared in 2 ways.

→ Assignment vs declarative.

```
printName();
var printName = function (name) {
    console.log(name);
}
```

(1)

```
printName1();
function printName1(name) {
    console.log(name);
}
```

(2)

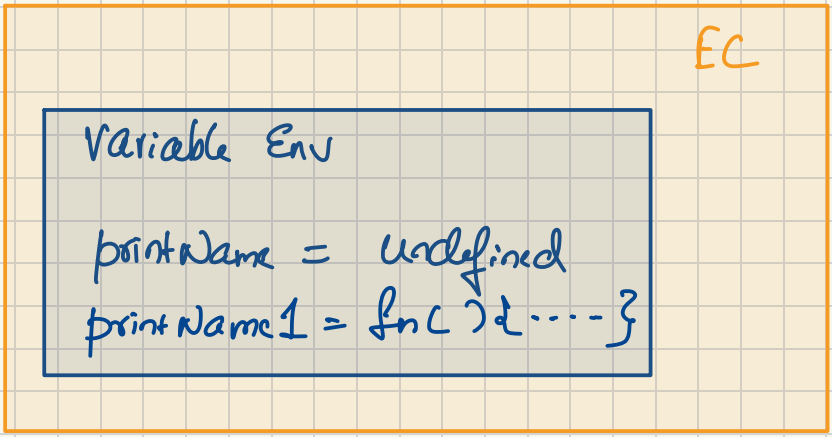
(1) ⟶ what is the o/p of 1?

The output of 1 is "printName is not a function. why?

(2) ⟶ what is the o/p of 2?

It prints the console log as expected.

Lets understand with the compiling phase:



Since printName is a assignment, it gets assigned to undefined just like a variable would in the compilation phase.

But since printName1 is a declaration, it is stored ; i.e the actual code is stored during compilation phase itself, hence during Execution it actually has the complete fn ready to execute.

The printName function only gets its assignment i.e. the code for the actual function only during the execution phase unlike the declarative function printName1

Also the functions are stored in the HEAP, but accessed by the variable env;, so its actually in the HEAP, not the variable env; itself

Lets Consider the code below:

```
showNumber();

function showNumber() {
    console.log("I'm a number");
}

var showNumber = function() {
    console.log("Hello");
}
```

(Here the o/p one would expect is undefined but it is actually "I'm a newbie")

Why?

→ When a fnc) and a variable have the same name, the variable declaration is ignored at the compilation step. In other words, the fnc) takes priority.

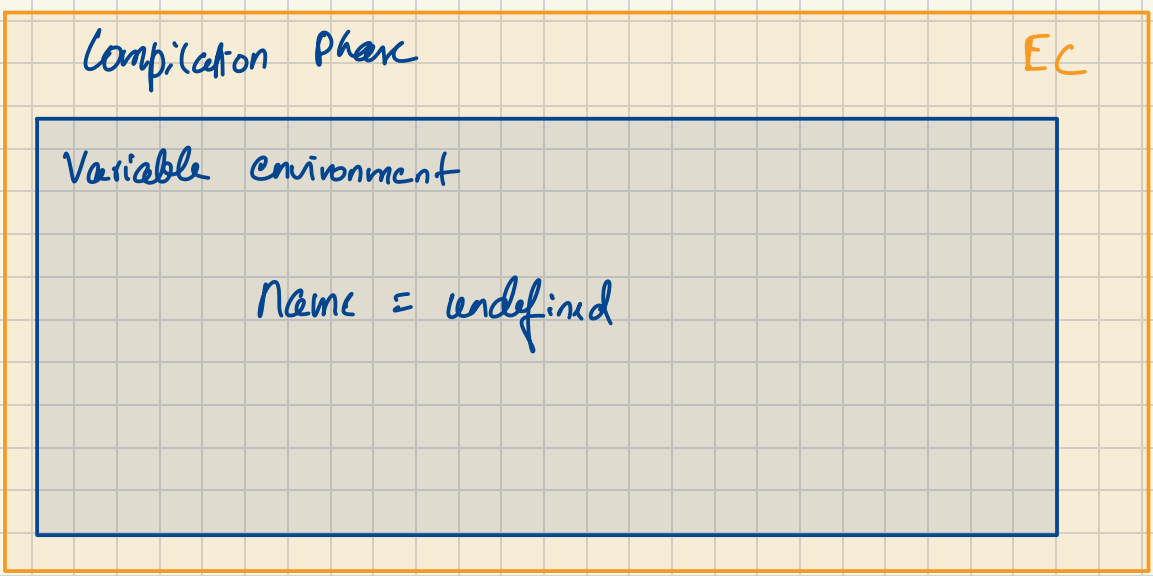
So, always ensure to unique names as much as possible.

```
console.log(name);
if(0){
    var name = "Shrikanth";
}
```

Here the if condition will never execute because 0 is a falsy value.

But lets take a look under the hood;

During the compilation phase, irrespective of whether the condition is truthy or falsy, variables are created. Hence in the EC ;-



If condition only applies to the execution phase.