JAVASCRIPT FUNCTION PARAMETERS AND ARGUMENTS

A JavaScript function does not perform any checking on parameter values (arguments)

- Function parameters are the names listed in the function definition.
- Functions arguments are the real values passed to (and received by) the function.

```
2   function fn(param1, param2) {
3       console.log("Inside fn", param1, param2);
4   }
5
6   fn("Hello", "Hi");
7   // if there is something missing then default is undefined
8   fn("Hello");
9   fn();
10  fn("Hello", "Hi", "By")
11
```

↑ Default Parameter: if a function is called with missing argument (less than declared), the missing value to the parameter.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

→  Lecture_41 git:(main) ✗ node 1_Function_Arguments.js
Inside fn Hello Hi
Inside fn Hello undefined
Inside fn undefined undefined
Inside fn Hello Hi
→  Lecture_41 git:(main) ✗
```

THE ARGUMENTS OBJECT

- JavaScript functions have a built-in object called the arguments object.
- The argument object contains an array of arguments used when the function was called (invoked).
- If a function is called with too many arguments (more than declared), these arguments can be reached using the arguments object.

```
18  function fn2(){
19      console.log(arguments);
20  }
21
22  fn2("Hello", "Hi");
23  fn2("Hello");
24  fn2();
25  fn2("Hello", "Hi", "By")
26
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

→  Lecture_41 git:(main) ✗ node 1_Function_Arguments.js
[Arguments] { '0': 'Hello', '1': 'Hi' }
[Arguments] { '0': 'Hello' }
[Arguments] {}
[Arguments] { '0': 'Hello', '1': 'Hi', '2': 'By' }
→  Lecture_41 git:(main) ✗
```

```
18    function fn2(param1, param2){
19        console.log(arguments);
20        console.log(param1, param2);
21    }
22
23    fn2("Hello", "Hi");
24    fn2("Hello");
25    fn2();
26    fn2("Hello", "Hi", "By")
27
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_41 git:(main) ✗ node 1_Function_Arguments.js
[Arguments] { '0': 'Hello', '1': 'Hi' }
Hello Hi
[Arguments] { '0': 'Hello' }
Hello undefined
[Arguments] {}
undefined undefined
[Arguments] { '0': 'Hello', '1': 'Hi', '2': 'By' }
Hello Hi
→  Lecture_41 git:(main) ✗
```

ARGUMENTS ARE PASSED BY VALUE

- The parameters, in a function call, are the function's arguments.
- JavaScript arguments are passed by value: The function only gets to know the values, not the arguments' locations.
- If a function changes an argument's value, it does not change the parameter's original value.
- Changes to arguments are not visible (reflected) outside the function.

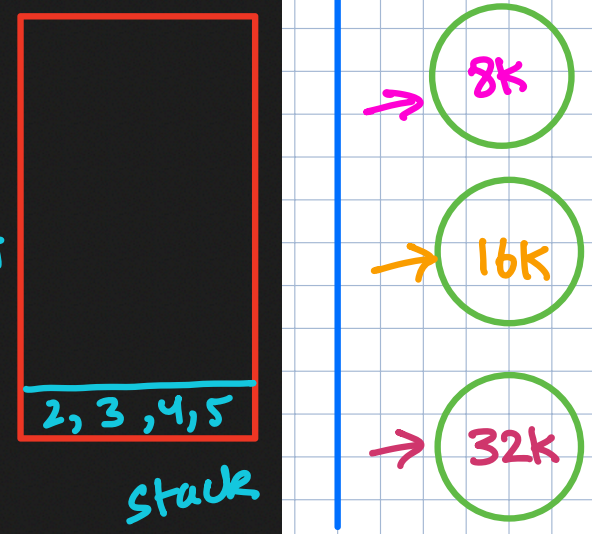
OBJECTS ARE PASSED BY REFERENCE

- In JavaScript, object references are values.
- Because of this, objects will behave like they are passed by reference:
- If a function changes an object property, it changes the original value.
- Changes to object properties are visible (reflected) outside the function.

```
Ques→ // Find the output of the following :-

function f() {
    console.log(arguments);
}

function f(a, b) {
    return a + b;
}

console.log(f(2, 3, 4, 5)); 14

function f(x, y, z, t) {
    return x + y + z + t;
}

console.log(f(2, 3, 4, 5)); 14
```

At the beginning, memory allocation happens.

f = 8K ~~16K~~ 32K



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_41 git:(main) ✗ node 2_Ques1.js
14
14
→  Lecture_41 git:(main) ✗
```

CODE EXECUTION

Code Execution has two levels :- → ① MEMORY ALLOCATION, ② CODE EXECUTION

① Memory allocation
- ① function definition → memory allocation is done
- ② variables → undefined set

② Code Execution

The code execution happens inside a bubble and that bubble is known as Execution Context. Execution context (EC) is defined as the environment in which the JavaScript code is executed. By environment we mean the value of "this", variables, objects and functions Javascript code has access to at a particular time.

The default execution context is known as Global execution context. (GEC) in which JS code start its execution when the file first loads in the browser. All of the global code i.e code which is not inside any function or object is executed inside the global execution context. GEC cannot be more than one because only one global environment is possible for JS code execution as the JS engine is single threaded.

Functional execution context (FEC): Functional execution context is defined as the context created by the JS engine whenever it finds any function call. Each function has its own execution context. It can be more than one. Functional execution context has access to all the code of the global execution context though vice versa is not applicable. While executing the global execution context code, if JS engine finds a function call, it creates a new functional execution context for that function. In the browser context, if the code is executing in 'strict' mode value of 'this' is "undefined" else it is window object in the function execution context.

VAR, LET and CONST

Global code

```
3   console.log("Before declaration", a);
4   var a;
5   console.log("After declaration", a);
6   a = 10;
7   console.log("After initialization", a);
8
```

undefined

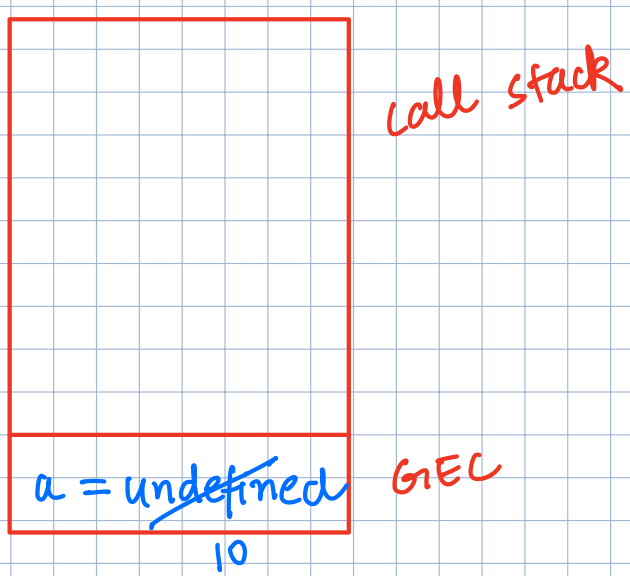
undefined

10

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_41 git:(main) ✗ node 3_Var_Let_Const.js
Before declaration undefined
After declaration undefined
After initialization 10
→  Lecture_41 git:(main) ✗
```

All the code that falls outside function runs under Global Execution Context. Variable are set undefined by default.



```
2  // G.E.C
3  // console.log("Before declaration", a);
4  var a;
5  // console.log("After declaration", a);
6  a = 10;
7  // console.log("After initialization", a);
8
9  function fn() {
10
11     console.log("Before declaration", a);
12
13     var a;
14
15     console.log("After declaration", a);
16
17     a = 20;
18
19     console.log("After initialization", a);
20
21
22 }
23
24 fn();
```

uf → 10

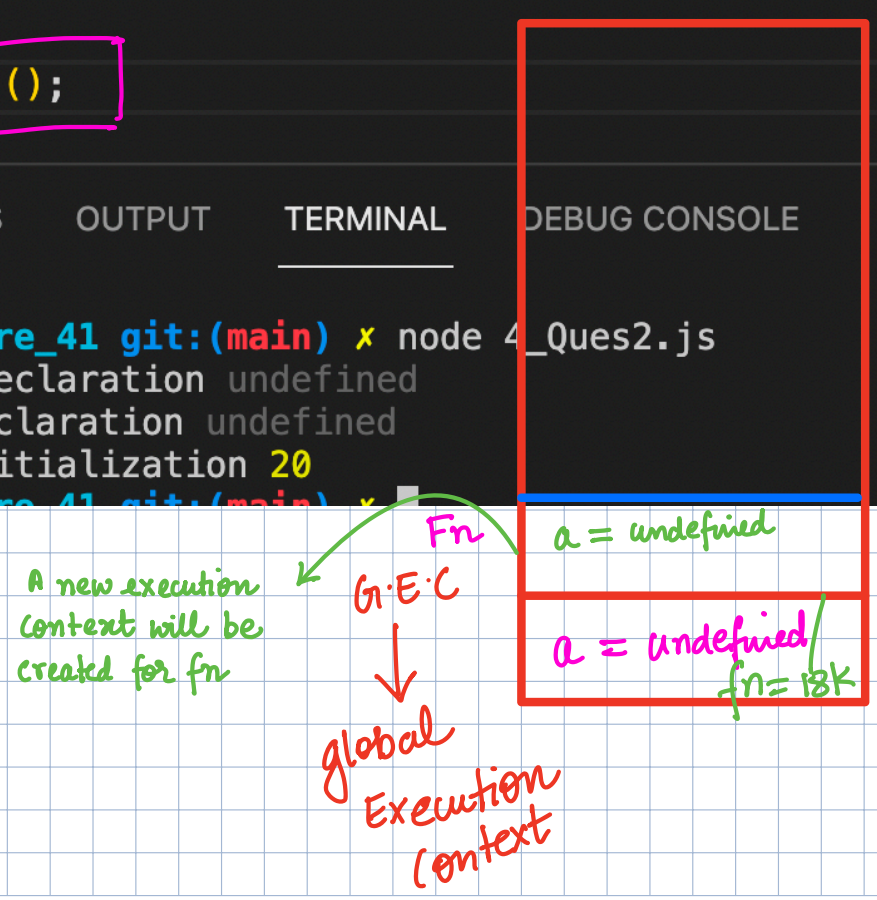
undefined

undefined

20

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_41 git:(main) ✗ node 4_Ques2.js
Before declaration undefined
After declaration undefined
After initialization 20
```



```
2   // G.E.C
3   console.log("Before declaration 3", a);
4   var a;
5   console.log("After declaration 5", a);
6   a = 10;
7   console.log("After initialization 7", a);
8
9   function fn() {
10
11      console.log("Before declaration 11", a);
12
13      var a;
14
15      console.log("After declaration 15", a);
16
17      a = 20;
18
19      console.log("After initialization 19", a);
20
21
22  }
23
24  fn();
```

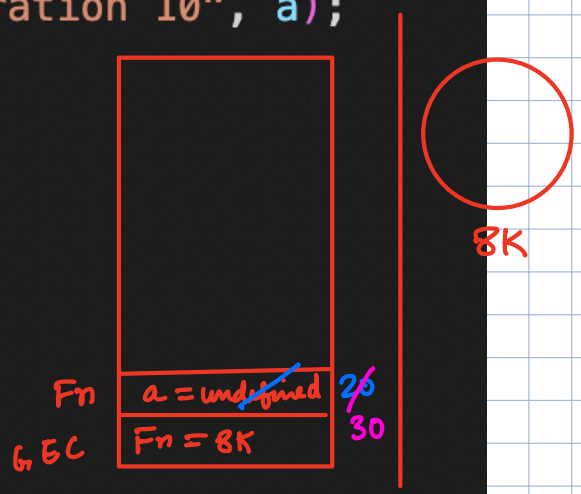
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_41 git:(main) ✗ node 4_Ques2.js
Before declaration 3 undefined
After declaration 5 undefined
After initialization 7 10
Before declaration 11 undefined
After declaration 15 undefined
After initialization 19 20
→  Lecture_41 git:(main) ✗
```

VAR and its Scope

```
2   // next question
3    // var is function scope.
4   function fn() {
5
6       console.log("Before declaration 6", a);
7
8       var a;
9
10      console.log("After declaration 10", a);
11
12      a = 20;
13
14      if(true) {
15          var a = 30;
16          console.log("16", a);
17      }
18
19      console.log("After initialization 19", a);
20
21  }
22
23  fn();
```

function scope



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_41 git:(main) ✗ node 5_Var_N_Scope.js
Before declaration 6 undefined
After declaration 10 undefined
16 30
After initialization 19 30
→  Lecture_41 git:(main) ✗
```

VAR V/S LET

```
1
2   // reassign
3   // function scope , redeclare, you can access a var variable before initialization
4
5   console.log(a);
6   var a;
7   a = 10;
8   var a;
9   console.log(a);
10
```

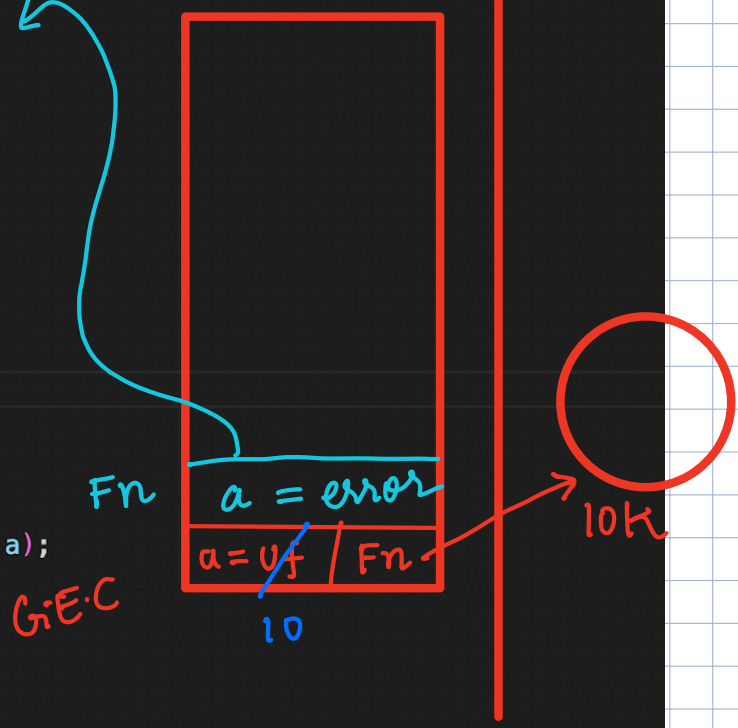
```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

➜  Lecture_41 git:(main) ✗ node 6_Var_Vs_Let.js
undefined
10
➜  Lecture_41 git:(main) ✗
```

```
2   // reassign
3   // function scope , redeclare, you can access a var variable before declaration
4
5   // console.log(a);
6   // var a;
7   // a = 10;
8   // var a;
9   // console.log(a);
10
11  // Before declaration -> for a variable declared using let keyword, that is in TDZ (Temporal Dead Zone).
12  // We cannot access the variable before declaration by let keyword. - > TDZ
13  // You can't redeclare a variable using let keyword
14  // Let keyword if block scope
15  // These things makes 'let' a safer options than 'var'
16
17  let a;
18  console.log(a);
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  zsh - Lecture

➜  Lecture_41 git:(main) ✗ node 6_Var_Vs_Let.js
undefined
➜  Lecture_41 git:(main) ✗
```

```
3   // next question
4   var a;
5
6   function fn() {
7
8       console.log("Before declaration 8", a);
9
10      let a;
11
12      console.log("After declaration 12", a);
13
14      a = 20;
15
16      if(true) {
17          let a = 30;
18          console.log("16", a);
19      }
20
21      console.log("After initialization 21", a);
22
23  }
24
25  a = 10;
26
27  fn();
```

why ? TDZ (Temporal Dead zone)



PROBLEMS OUTPUT TERMINAL DEBUG CONS

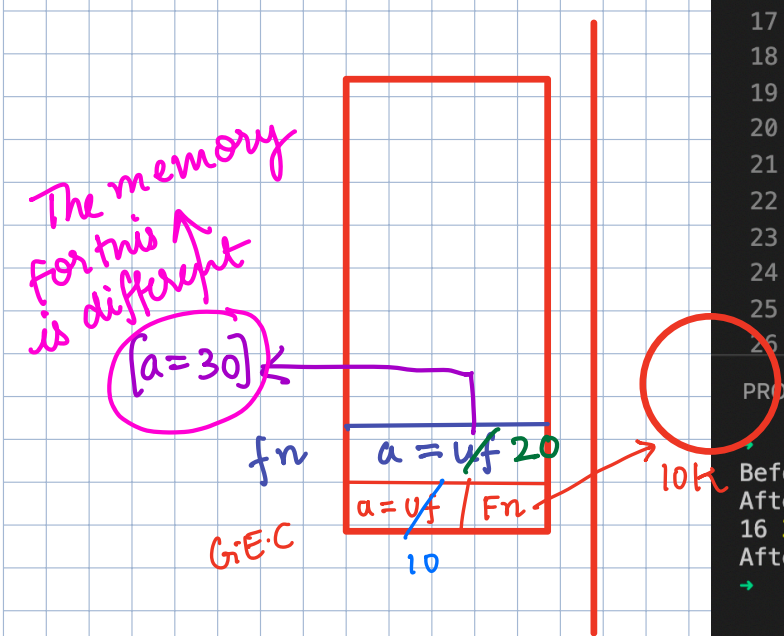
```
→  Lecture_41 git:(main) ✗ node 7_Ques3.js
/Users/hemakshipandey/Desktop/Pepcoding-FJ
    console.log("Before declaration 8", a)
                                        ^

ReferenceError: Cannot access 'a' before i
    at fn (/Users/hemakshipandey/Desktop/P
    at Object.<anonymous> (/Users/hemakshi
    at Module._compile (internal/modules/c
```

```
3   // next question
4   var a;
5
6   function fn() {
7
8       console.log("Before declaration 8", a);
9
10      var a;
11
12      console.log("After declaration 12", a);
13
14      a = 20;
15
16      if(true) {
17          let a = 30;
18          console.log("16", a);
19      }
20
21      console.log("After initialization 21", a);
22
23  }
24
25  a = 10;
```

undefined

undefined

30



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_41 git:(main) ✗ node 7_Ques3.js
Before declaration 8 undefined
After declaration 12 undefined
16 30
After initialization 21 20
→  Lecture_41 git:(main) ✗
```

DIFFERENCE BETWEEN VAR, LET AND CONST

	var	let	const
origins	pre ES2015	ES2015(ES6)	ES2015(ES6)
scope	globally scoped OR **function** scoped. attached to window object	globally scoped OR **block** scoped	globally scoped OR **block** scoped
global scope	is attached to Window object.	not attached to Window object.	attached to Window object.
hoisting	**var** is hoisted to top of its execution (either global or function) and initialized as ***undefined***	**let** is hoisted to top of its execution (either global or block) and left **uninitialized**	**const** is hoisted to top of its execution (either global or block) and left **uninitialized**
redeclaration within scope	**yes**	**no**	**no**
reassigned within scope	**yes**	**yes**	**no**

WHAT IS TEMPORAL DEAD ZONE?

let and const have two broad differences from var:

1. They are block scoped.
2. Accessing a var before it is declared has the result undefined; accessing a let or const before it is declared throws ReferenceError:

```
console.log(aVar); // undefined
console.log(aLet); // Causes ReferenceError: Cannot access 'aLet' before initialization

var aVar = 1;
let aLet = 2
```

It appears from these examples that let declarations (and const, which works the same way) may not be hoisted, since aLet does not appear to exist before it is assigned a value.

That is not the case, however—let and const are hoisted (like var, class and function), but there is a period between entering scope and being declared where they cannot be accessed. This period is the temporal dead zone (TDZ).

The TDZ ends when aLet is declared, rather than assigned:

```
// console.log(aLet) // Would throw ReferenceError

let aLet;

console.log(aLet); // undefined
aLet = 10;
console.log(aLet); // 10
```

This example shows that let is hoisted:

```
let x = "outer value";

(function() {
  // Start TDZ for x.
  console.log(x);
  let x = "inner value"; // Declaration ends TDZ for x.
}());
```

Credit: http://jsrocks.org/2015/01/temporal-dead-zone-tdz-demystified

Accessing x in the inner scope still causes a ReferenceError. If let were not hoisted, it would log outer value.

The TDZ is a good thing because it helps to highlight bugs—accessing a value before it has been declared is rarely intentional.

The TDZ also applies to default function arguments. Arguments are evaluated left to right, and each argument is in the TDZ until it is assigned:

```
// b is in TDZ until its value is assigned.
function testDefaults(a = b, b) { }

testDefaults(undefined, 1); // Throws ReferenceError because the evaluation of a reads b before it has been evaluated.
```

The TDZ is not enabled by default in the babel.js transpiler. Turn on "high compliance" mode to use it in the REPL. Supply the es6.spec.blockScoping flag to use it with the CLI or as a library.

Recommended further reading: TDZ demystified and ES6 Let, Const and the “Temporal Dead Zone” (TDZ) in Depth