

```
const cutPieces = function (fruit) {  
  return fruit * 4;  
};  
  
const fruitProcessor = function (apples, oranges) {  
  const applePieces = cutPieces(apples);  
  const orangePieces = cutPieces(oranges);  
  
  const juice = `Juice with {applePieces} pieces of  
apple and ${orangePieces} pieces of orange.`;  
  return juice;  
};  
  
console.log(fruitProcessor(2, 3));
```

The diagram illustrates the execution flow of the provided JavaScript code. Red arrows represent function calls and return values, while a yellow arrow represents the final argument passed to the console.log function.

- cutPieces function:** Takes a parameter `fruit` and returns `fruit * 4`.
 - Call 1: `cutPieces(apples)` where `apples = 2`. Returns `8`.
 - Call 2: `cutPieces(oranges)` where `oranges = 3`. Returns `12`.
- fruitProcessor function:** Takes parameters `apples` and `oranges`.
 - It calls `cutPieces(apples)` (returns `8`) and `cutPieces(oranges)` (returns `12`).
 - It constructs a string `juice`Juice with {applePieces} pieces of apple and ${orangePieces} pieces of orange.`` where `applePieces` is `8` and `orangePieces` is `12`.
 - It returns the constructed string.
- console.log:** Calls `fruitProcessor(2, 3)`. The arguments `2` and `3` are highlighted with red and blue boxes respectively.

* functions : (Abstraction \rightarrow hiding Implementation details)

- \rightarrow Solves the problem of DRY, provides reusability
- \rightarrow Can take Inputs in form arguments
- \rightarrow Can give outputs/result using return keyword
- \rightarrow return shutdowns the function, any after return will not be executed, you can send any values with return.
- \rightarrow we used many functions without knowing them,

const a = Number("123")

const s = String(123)

const b = parseInt(5/10)

fromCharCode ()

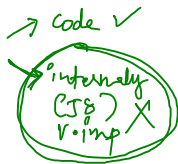
charAt ()

console.log ()

What ?

how to use ?

how they work



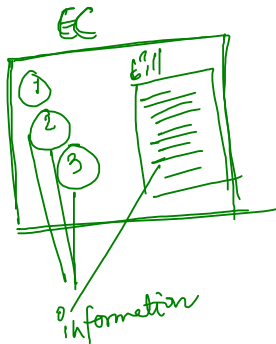
9:15pm - 9:30pm

BREAK

* Execution Context :

→ It is like a box, where a piece of JS code is executed
It contains all info for the code to be executed

- 1. shoes
- 2. earphones
- 3. iPhone



- 1. variables
 - let, const, var
 - function
 - parameter / arguments

2. scope chain ✗

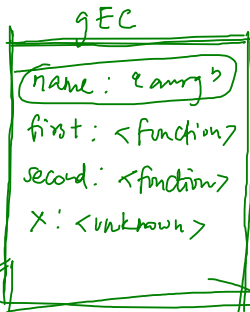
- only one EC is created per program
- EC is created for every function

```

530 const name = "anurag";
531
532 function first() {
533   let a = 1;
534   const b = second();
535   a = a + b;
536   return a;
537 }
538
539 function second() {
540   var c = 2;
541   return c;
542 }
543
544 const x = first();
545 console.log(x);

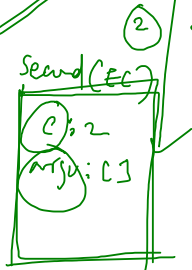
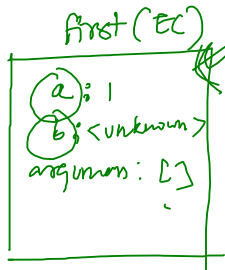
```

① A global Execution Context is created

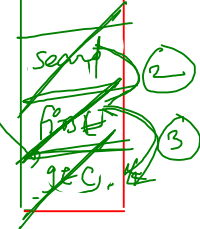


* whichever EC is there on top of stack that will run

JS (V8) engine



② run the gEC
→ gEC is placed on to stack



(call stack) [platform]

↓ 500k

Global scope

```
const a = 5;  
let fname = "arjun";  
let year = 2023;
```

- outside of a function or block
- they are accessible anywhere in the code.

function scope

```
function calcAge(year){  
  year = year + 1;  
  const age = 2023 - year;  
  return age;  
}  
  
const year = 2000;  
console.log(calcAge(year));  
console.log(year);  
console.log(age);
```

reference error

- variables are accessible inside the function only.
- also called as local scope.

block scope

```
577 let year = 1992;  
578 if (year >= 1981 && year <= 1996) {  
579   const oldGen = true;  
580 }  
581 console.log(oldGen);
```

reference error

- variables are accessible only inside block (if, else, else if, for, while, ~~do while~~)
- this applies only to let and const variables (Twist)
- var is only function scope and not block scoped due to hoisting.