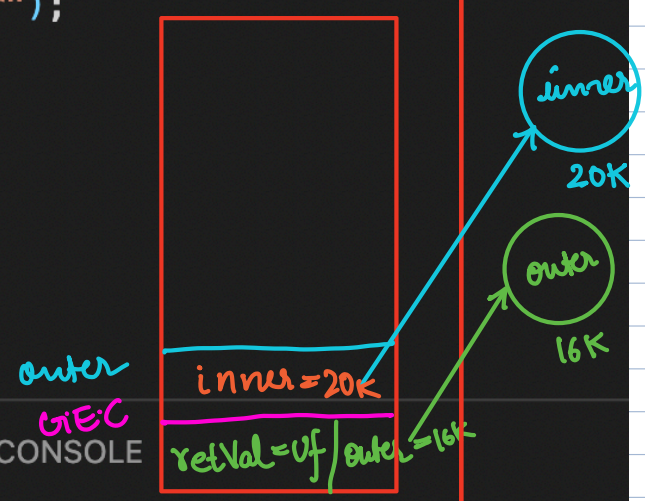
# FUNCTIONS

```
// functions are variable

function outer(param){
    console.log(param);
    param();
}
function chotaFn(){
    console.log("Hello I am a chota fn");
}
// 1. function can be passed as a paramenter to another to another function -> higher Order function
outer(chotaFn);

// 2. function's reference can be stored in another variable -> function expression
let a = [1, 2, 3, 4, 5];
let b = a;
//function expression
let anotherName = function(){
    console.log("I am an expression");
}
anotherName();

// 3. A Variable can be returned by function, similarly, a function can also be returned by a function.
function fn() {
 return "Hello";
}
let rval = fn();
console.log(rval)
```

```
function outer() {
    console.log("Hello i am outer and i am returning Inner");
    return function inner() {
        console.log(" I am Inner");
    }
}

let retVal = outer();
console.log(retVal);
```



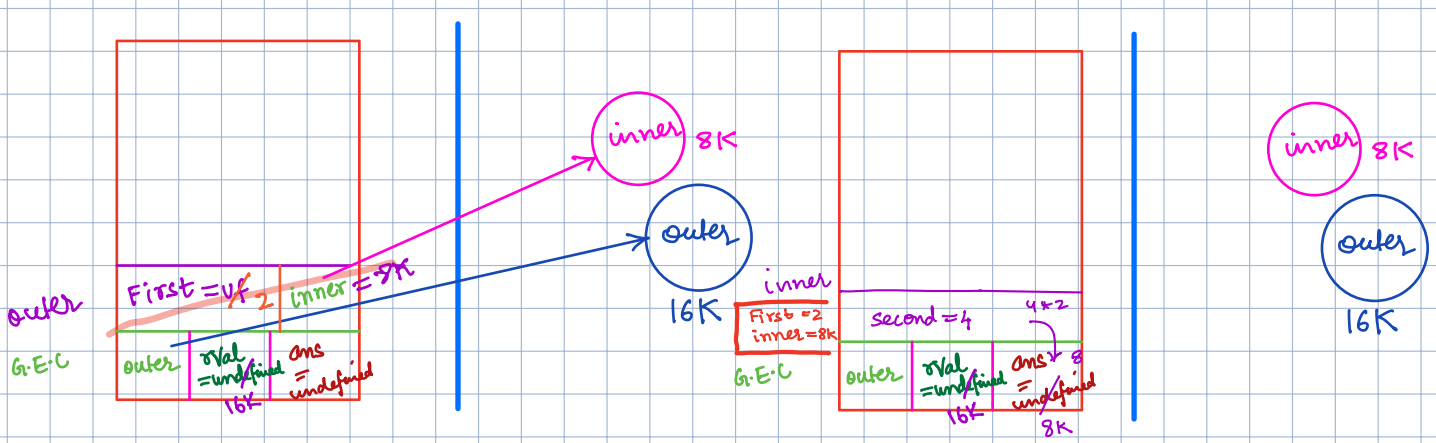

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

→  Lecture_42 git:(main) ✗ node 1_function.js
Hello i am outer and i am returning Inner
[Function: inner]
→  Lecture_42 git:(main) ✗
```

# CLOSURE

```
1
2  function outer(first) {
3      console.log(" Inside outer ");
4      return function inner(second){
5          console.log("Inside inner");
6          return first*second;
7      }
8  }
9
10 let rVal = outer(2);
11 console.log("Before calling rVal that contains inner");
12 let ans = rVal(4);
13 console.log(ans);
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_42 git:(main) ✗ node 2_closures.js
 Inside outer
Before calling rVal that contains inner
Inside inner
8
→  Lecture_42 git:(main) ✗
```



# what is closure?

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.

Closure means that an inner function alway has access to the vars and parameters of its outer function, even after the outer function has returned.

Consider the following code example:

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```

Running this code has exactly the same effect as the previous example of the `init()` function above. What's different (and interesting) is that the `displayName()` inner function is returned from the outer function *before being executed*.

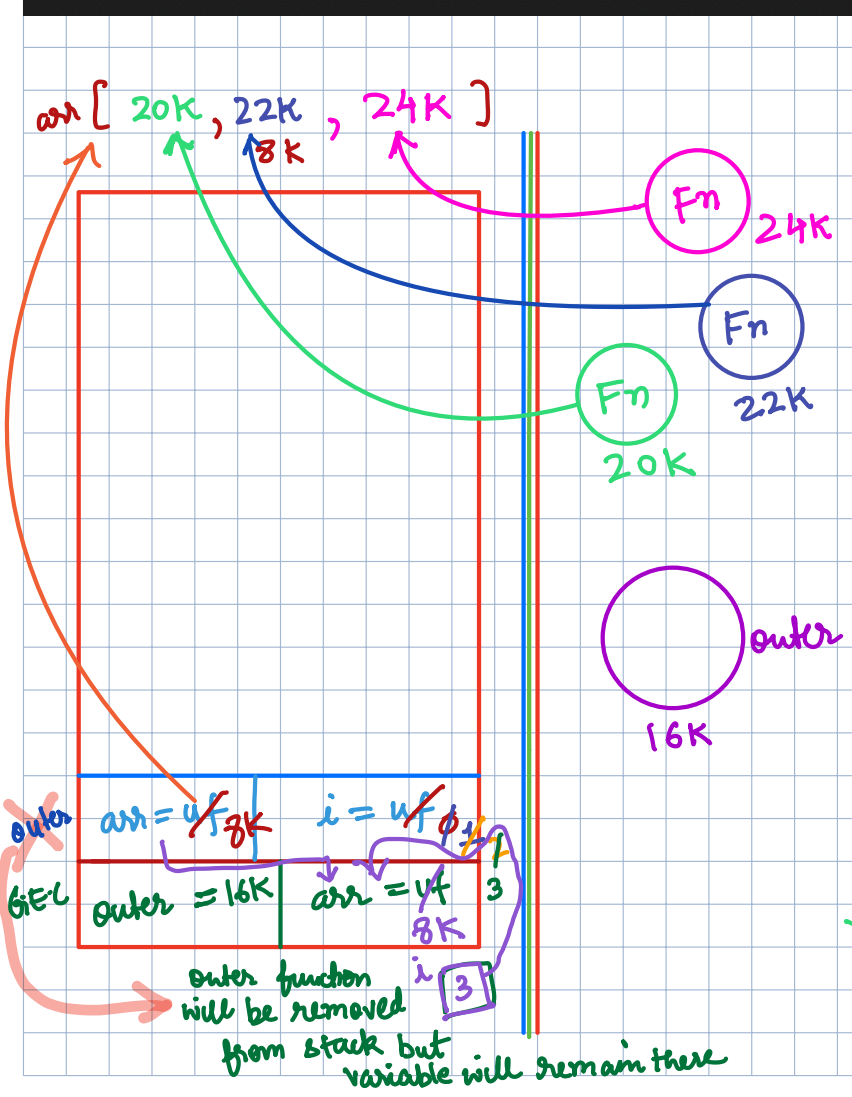
At first glance, it might seem unintuitive that this code still works. In some programming languages, the local variables within a function exist for just the duration of that function's execution. Once `makeFunc()` finishes executing, you might expect that the name variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created. In this case, `myFunc` is a reference to the instance of the function `displayName` that is created when `makeFunc` is run. The instance of `displayName` maintains a reference to its lexical environment, within which the variable `name` exists. For this reason, when `myFunc` is invoked, the variable `name` remains available for use, and "Mozilla" is passed to `alert`.

```
function enterFirstName(firstName) {
    return function enterLastName(lastName){
        return function enterAge(age) {
            return function printDetails(){
                console.log("Your name is " + firstName + " " + lastName + " and your age is " + age);
            }
        }
    }
}

console.log("Kindly Enter your first Name");
let enterLN = enterFirstName("Hemakshi");
console.log("Kindly Enter your last Name");
let enterA = enterLN("Pandey");
console.log("Kindly Enter your age");
let printDT = enterA(15);
console.log("Doing random Stuff");
printDT();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE zsh - Lectur

```
→  Lecture_42 git:(main) ✗ node 3_closures.js
Kindly Enter your first Name
Kindly Enter your last Name
Kindly Enter your age
Doing random Stuff
Your name is Hemakshi Pandey and your age is 15
→  Lecture_42 git:(main) ✗
```




```
function outer(){
    var arr = [];
    for(var i = 0; i < 3; i++){
        arr.push(function () {
            console.log(i);
        })
    }
    return arr;
}

console.log("Before calling order");
var arr =  outer();
arr[0]();
arr[1]();
arr[2]();
console.log("Aftercalling outer")
```

inside array we are pushing a function

8K

arr[0](); → access value of i → 3

arr[1](); → access value of i → 3

arr[2](); → access value of i → 3

obtains from closure

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
→  Lecture_42 git:(main) ✗ node 4_Ques1.js
Before calling order
3
3
3
Aftercalling outer
→  Lecture_42 git:(main) ✗
```

```
// print 1, 2, 3 instead of 3, 3, 3

// solution1 -> Execution Context
function outer(){
    var arr = [];
    for(var i = 0; i < 3; i++){
        function outer1(){
            var j = i;
            return function () {
                console.log(j);
            }
        }
        arr.push(outer1());
    }
    return arr;
}

console.log("Before calling order");
var arr =  outer();
arr[0]();
arr[1]();
arr[2]();
console.log("After calling outer")
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOL

```
→  Lecture_42 git:(main) ✗ node 5_Ques2.js
Before calling order
0
1
2
After calling outer
```