Closures :

A closure in JS is a function that "remembers" the variables from its outer scope. even after the outer function has finished execution.

① Data privacy & Encapsulation :-

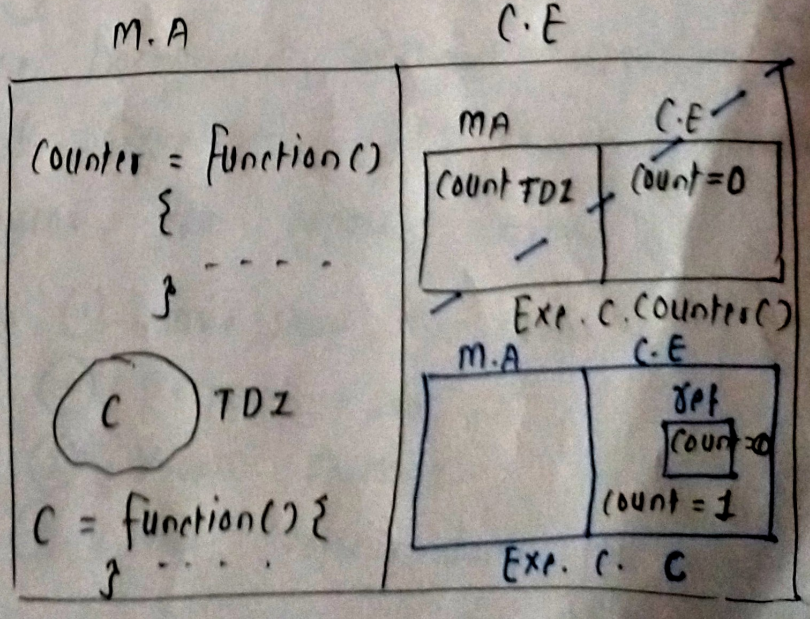
```
function Counter() {
    let count = 0 ;    // private variable.
    return {
        increment() {
            count++;
            console.log(count);
        },
        decrement() {
            count --;
            console.log(count);
        }
    };
}
let counter_fun = Counter();
counter_fun.increment(); // 1
```

```
function counter (){
    let count = 0;   // private variables.
    return function (){
        count++;
        console.log (count);
    }
};

let c = counter ();
```

so in c we have `function (){ count++; console.log (count); }`

i.e

```
c = function (){
    ---
}

c();  // 1
c();  // 2
c();  // 3
```



* Synchronous and Asynchronous programming :-

(1) Synchronous programming :- In this tasks are executed One at a time, in the Order they appear.
The next task Starts Only after the Current One is Completed.

Example :- Imagine you are in coffee shop, and there is Only One barista (coffee Server). The barista takes One Order, makes the coffee & Serve it before taking the Next Order. (Petrol pump).

Code Example :-

```
console.log ("Order coffee")
console.log ("Make coffee")
console.log ("Serve coffee")
```

Here, each task waits for the previous One to Complete before starting.

(2) ~~Asy~~ Asynchronous programming :- In this, tasks can start & continue running in the background, without waiting for each other. When a task is completed, it's result is handled.
Imagine you give your laptop for repair, insted of waiting at the shop : you (1) Leave shop, go home
(2) prepare Food
(3) Wash Clothes
(4) Once repair done, the shop calls you.

Code Example :-

① → console.log ("Give laptop for repair");

setTimeOut (() => { console.log ("Laptop repair done. pick up!"); ④

}, 3000);

② → console.log ("start preparing Food");

③ → console.log ("start washing Clothes");

The repair task runs at background & other tasks continue immediately.

---

*

Callback hell :- Callback hell occurs when multiple asynchronous tasks are dependent on each other. & nesting of callbacks becomes to deep & hard to manage.

Code Example :- (such types of code you can write only when you practice & revise).

```
function placeOrder (item, callback) {
    setTimeOut (() => {
        console.log ("Order placed, Pyment progress");
        callback ();
    });
};

function makepayment (amt, callback) {
    setTimeOut (() => {
        console.log ("Payment Success of Amt", amt);
    })
};

function confirmOrder (callback) {
    setTimeOut (() => { console.log ("Order Confirmed");
        callback(); });
};

function SendEmail () {
    setTimeout (() => { console.log ("Email Sent");
});
```

// Calling Functions :-

```
placeOrder ("pizza", () => {
    makepayment (2500, () => {
        confirmOrder (() => {
            sendEmail ()
        });
    });
});
```

Nested Callback (Callback Hell).
Pyramid of Doom.

: Promises : solution of Callback Hell.

Promises are objects that represents the eventual Completion Or failure of an Asynchronous Operation.

resolve, reject → callbacks provided by JS.
Handlers.

States :- (1) Pending
(2) resolved
(3) reject.

Ex : (1)

```
let p1 = new Promise (resolve, reject) {
    console.log ("I am promise");
    reject ("Some Error Occured");
});
```

```
let p2 = new Promise (resolve, reject) {
    reject ("some Error"); // finally ("done");
});
```

```
let p3 = new promise (resolve, reject) {
    resolve ("Success");
});
```

: then() & ~~catch()~~ : catch()

```
let p1 = new Promise ((resolve, reject) {
    setTimeOut (() => {
        console.log ("Data fetched");
    }); resolve();
```

↑ start of this promise is resolved so Now call .then method.

```
p1 . then (function processData () {
    }); console.log ("Data process started");
```

↑

if it reject ⟶ catch will be called.

Ex: (2)

```
let p1 = new promise ((resolve, reject) => {
    setTimeout (() => {
        resolve ("success"); / reject ("failed");
    });
});

p1.then (function (res) {
    console.log (res);
}).catch (function (res) {
    console.log (res)
});
```

Promise Chaining (Solution to callback hell) :-

```
function placeOrder () {
    return new Promise ((resolve, reject) => {
        setTimeout (() => {
            console.log ("Order placed");
            resolve ();
        }, 1000);
    });
}

function makePayment () {
    return new Promise ((resolve, reject) => {
        setTimeout (() => {
            console.log ("Payment success");
            resolve ();
        }, 2000);
    });
}

function confirmOrder () {
    return new promise ((resolve, reject) => {
        setTimeout (() => {
            console.log ("Order placed /confirmed");
        }, 1000);
    });
}

function emailSent () {
    return new promise ((resolve, reject) => {
        setTimeOut (() => {
            console.log ("Email sent successfully");
        }, 1000)
    });
};
```

```
function ProcessOrder () {
    placeOrder ()
    . then (() => makepayment ())
    . then (() => confimOrder ())
    . then (() => emailSent ())
    . do
    . then (() => console.log ("Process End");
    . catch
    . catch (() => console.log ("Error :", er);
    console.log ("The process Completely Ended, Thank You");
};
```

```
async
try { function ProcessOrder () {
    await placeOrder ();
    await makepyment ();
    await confirmOrder ();
    await emailsent ();
}
catch (err) {
    : console.log ("Error", err);
}
console.log ("The process completed, Thank You");
}
```

async & await

```
demo();
function demo (){
    console.log ("Task1");  (1)
    p1. then (()=> console.log ("Task2"));  (4)
    console.log("Task3");  (2)
}

console. log (" Task 4");  (3)
```

```
async function demo (){
    console.log ("Task1");  (1) → (1)
    await p1 ;
    console.log ("Task 2");  (2) → (3)
    console.log ("Task 3");  (3) → (4)
}

console.log ("Task 4");  (4) → (2)
```

---

async :- defines asynchronous function.

await :- wait for promis to fullfill then only proceed with Next Task within funt° but when promise is under pending it to parallely executes instructions Outside the funt°.