✳ Asyncronity ✳

JS: The New Hard Parts

✳ to communicate with the outside world like getting data from server, (XHR) using Timeout. all these are not the JS feature but the web browser feature.

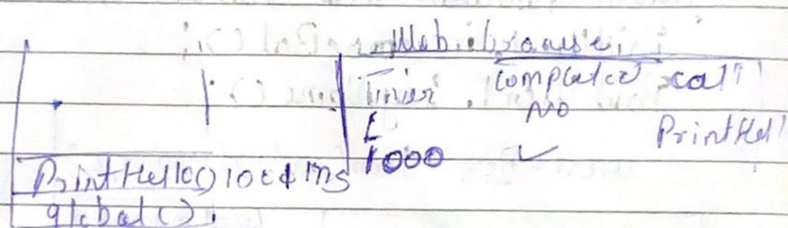
```
function printHello () {
    con.Log ("Hello" ); }
```

setTimeout (printHello, 1000)
↓ Web browser feature → Timer

```
setTimeOut ( printHello, 1000 );
c.L ("Hi")
```

2ms — c.L → Hi

Output :-
Hi
Hello



⇒ Whenever there is a Async code (i.e facade function) the JS first execute all the code in the global context. then it will check the event Loop (i.e) microtask queue & callback queue (priority is for microtask queue & then callback queue.

✳ Promises ✳

⇒ Using facade function

```
function display (data) { console.log (data); }
const futureData = fetch ("twitter.com/get/1");
futureData. then (display) ;
console. log ("Me first");
```

futureData = fetch ("twitter/get/1");
↓ In JS ↳ browser

Promise object
{value: }
{on fullfillment: [ ]}

futureData.then (display); onfullfillment ⟶ [display]

① console.Log ("me first")
C.L

Global memory
display : ƒ ②
futureData : { value:
onfullfillment
display : ƒ }

Fascade function
↳ fetch

② Web browser
1ms
get
XHR
200
url/endpoint
Get
completed
201 "Hi"
on Completion
futureData.value = "Hi" and also
sets the Status: resolved.

Whenever we do a fetch its a fascade function. i.e it mask 2 operation, 1) in JS and 2) in browser

JS → it creates a Promise object with the property Value, onfullfillment,

browser → it does the following :-
Creates a XHR request, & when the response is received from the request it updates the Promise.value, which will then triggred the callback function, which is passed in the .then

★ So there is event Loop that has link to the callback queue (Task queue) and micro task queue (Job queue), the JS gives the Priority to the Job queue than the callback queue, when we do Something that requires connection with (urls, request, response) then they go on to the microtask/Job queue.

So whenever global stack is completed its execution it checks the Microtask queue & then the callback queue, first all the microtask queue, functions are executed & then, callback queue,

* the Promise object has 3 properties,
  Status: Pending, resolved, rejected
  onfullfillment: callback function
  onError :: catch block
  ⇒ The callbackfunction is called on the Status, updating property.

So in case of error whatever we pass onto the second, argument is called,

—— Iterators ——

```
function createfun(){
    function add2(num){
        return num+2; }
    return add2; }

const nf = createfun();
const add = nf(2); // 4
```

nf = createfun()

	LM
	add2: f
return	

add = nf(2)

	LM
return 2+2	num = 2
4	

GM
createfun: f
nf: f (add2(num))
add: 4

The inner function defination is return

```
function nextEle(array){
  let i=0;
  function inner(){
    const element = array[i];
        i++;
        return element;}
    return inner;
}

const returnNxt = nextEle([3,4,5]);
const e1 = returnNxt(); //3
const e2 = returnNxt(); //4
```

returnNxt = nextEle([3,4,5]);

```
return inner;
```

LM
array: [3,4,5];
i = 0;
inner: f

e1 = returnNxt();

```
i=0   1
array[i]
[3,4,5][0]
```

LM
element: 3
return element

Global mem
nextEle: f
returnNxt: inner: f
closure
array: [3,4,5]
i = 0;
i = 1
e1: 3

☆ Using iterator object & next method:

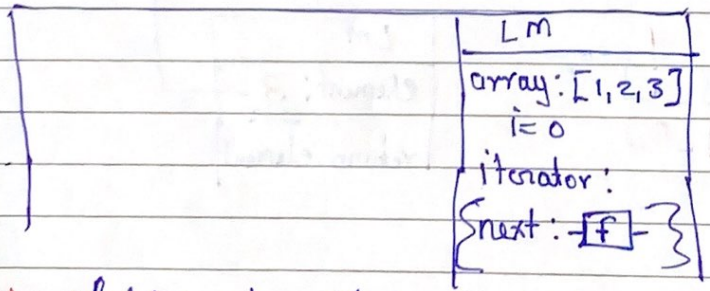
```
function createFlow(array){
  let i=0;
  const iterator = {
    next: function(){
      const ele = array[i];
      i++;
      return ele;
    }
  }
  return iterator; }

const returnNext = createFlow([1,2,3]);
const ele1 = returnNext.next();
const ele2 = returnNext.next();
```

GM
createFlow: f
returnNext:
{next: f}
array [1,2,3]
i = 1

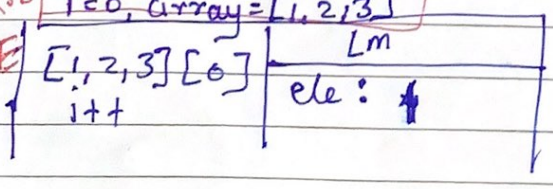
ele1: 1
ele2: 2

~~let~~ returnNext =



ele1 = returnNext.next()

closure
P.L.S.R.D
C.O.V.E



Generator function

```
function *createFlow(){
    yeild 4;
    yeild 5;
    yeild 6; }
const returnNextElement = createFlow()
const ele1 = returnNextElement.next()
const ele2 = returnNextElement.next()
```

memory

createFlow: { * f }

returnNextElement: { next: f }

ele1: 4

ele2: 5

returnNextElement = createFlow()

→ { next: f }

element1 = reNEL.next() ··· (createFlow()) (create bond

4

yeild 4	memory
suspends execution	

yeild 5

generator

The createFlow will create a object that has next(), method in it.

It's a ES7 feature

```
function *createflow( ) {
  const num = 10;
  const newNum = yeild num;
    yeild 5 + newNum;
    yeild 6;
}

const retNextEle = createflow()
const ele1 = retNextEle.next();
    ele2 = retNextEle.next(2);
```

createflow : ƒ

retNextEle : { next: ƒ }

ele1: 10

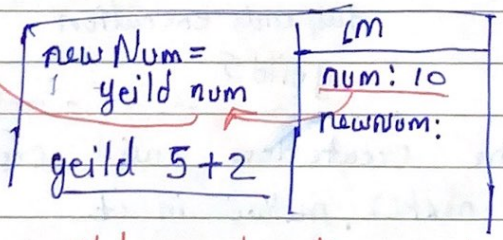
ele2: 7

retNextEle = createflow()

↳ { next: ƒ }

ele1 = retNextEle.next()

10 ← 10



whenever we do yeild and the assingment is there, it immediately jumps out of the function, without assinging, & after when we call .next() again with parameter what we pass as parameter is assigned to the (=) variable, where we stoped execution.

ele2 = retNextEle.next(2)

← 7

The generator does not finish the execution it stores current line, & scope in [[generator]]

Memory

```
function doWhenDataRec(data){
  retNextEle.next(data); }

function *createFlow(){
  const data = yeild fetch("get1,");
  console.log(data); }

const retNextEle = createFlow();
const getValue = retNextEle.next();
  value.then(doWhenDataRec);
```

doWhenDataRec : f

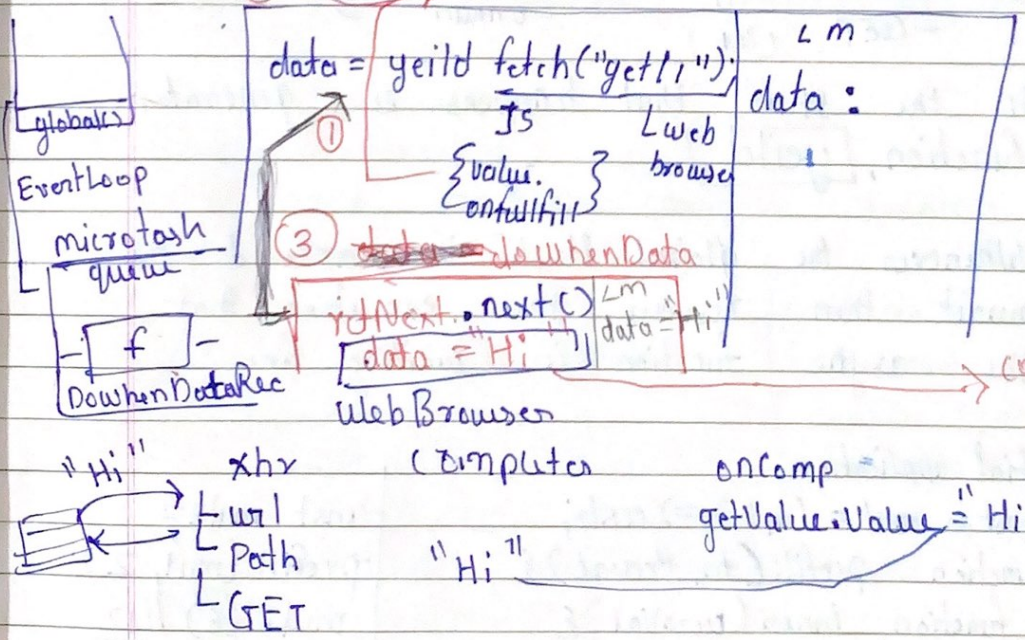
createFlow : * f

retNextEle : ① { next: f }

getValue : ② { value: "Hi", onfullfilling: }

retNextEle = createFlow()
└ { next: f }

getValue = retNextEle.next()



when the value is received back then we Execute doWhenDataRec, & then go back to the execution context of the createflow

Jake Archibald JS Speakes

Async Await

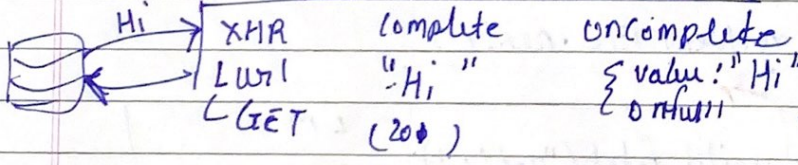
```
async function createFlow(){
  CL("me first");
  const data = await fetch("getHi");
  CL(data); }

createFlow();
CL("me second");
```

memory: createFlow: f

CL:
me first
me second
"Hi"

data = await fetch {value onFull} | data: Hi



All the stuff that happens is generator function, yeild

Whenever the global fetch is completed the await then resumes the execution that the async function is waiting for.

Partial application

```
const mul = (a,b) => a*b;
function prefill(fn, preVal){
  function inner(currVal){
    const op = fn(preVal, currVal);
    return op; }
  return inner; }
```

```
const mul2 =
prefill(mul, 2);
mul2(6) //12
```

★ function composition :- (best way to do more function Nesting)

```
Const   mul2 = num => num x 2;
cont    add3 = num => num + 3;

const reduce = (array, howtoJoin, buildup) {
  for (let i=0; i<array.length; i++) {
    buildup = howtoJoin(buildup, array[i]);
  }
  return buildup; }

                        inp, fn
const runOnInp = (~~fn, inp~~) =>
  { return fn(inp) }

~~const array = [mul2, add3];~~
const reduce = (~~fn~~ array, runOnInp, 10);
```

★ function Decoration :-

To control the inner function running to run only once.

```
function outer(convertMe) {
  let counter = 0;
  function inner(input) {
    if (counter === 0) {
      const output = convertMe(input);
      counter++;
      return output; }
    return "sorry"; }
  return inner;
}
```

```
const mul2 = num => num x 2;
const onMul = outer(mul);
onMul(10) // 20
onMul(20) // sorry
```