Class 7th

* Apply polyfill
* Bind polyfill
* flat polyfill

* Start with OOPs

⇒ call ⇒ functionName.call(obj, arg1, arg2, arg3);

⇒ Apply ⇒ functionName.apply(obj, argArray);

Function.prototype.call(objContext, ...restArg){

→ give me array of arguments

* objContext.tempFn = this;
* Const res = objContext.tempFn(...restArgs);
* delete objContex.tempFn;
* return res;

}

apply

Function.prototype.call(objContext, ...restArg) {

this rest operator is not required now

give me array of arguments

* objContext.tempFn = this;
* const res = objContext.tempFn(...restArgs);
* delete objContex.tempFn;
* return res;

}

function test() {

}

test();

test.fn();

objContext

objContext.tempFn = this

Apply

↳ Copy this key word

(will store fn defi. of with which it's called

fn temp

fn ABC

Bind ?

Some

* fn.call (obj, arg1, arg2, arg3);

* fn.call (obj, 1, 2, 3);

* fn.call (obj, 4, 5, 6);

---

* newFn (arg1, arg2, arg3);

it's a wrapper over fn.call (obj)

```
Const newFn = oldFn.bind(obj,          );
```

① we can pass arg in bind

② we can pass arg in new Fn

```
oldFn.bind(obj, 1);
```

```
* oldFn.call(obj, 1);
```

```
newFn(2, 4, 5);
```

```
oldFn.call(obj, 1, 2, 4, 5);
```

* any arg that is passed in bind method while creating a new Fn, will fix those arg in call method.

---

* new Fn → oldFn.call(obj,)

this syntax is seeking args

* newFn = oldFn.bind(obj);

↳ newFn();

↳ oldFn.call(obj);

this syntax is seeking args

---

oldFn → accepts 3 args → arg1, arg2, arg3

* newFn(1, 2, 3, 4) ⇒

↳ oldFn.call(obj, 1, 2, 3, 4);

* newFn = oldFn.bind(obj, 1);

↳ oldFn.call(obj, 1); → fn is still seeking args

⇒ newFn(4, 5, 6)

↳ oldFn.call(obj, 1, 4, 5, 6);

```
Function.prototype.Bind = function (objContext, ... fixedArgs) {
    const oldFn = this;
    const newFn = function (...nonFixedArgs) {
        return oldFn.call(objContext, ...fixedArg, ...nonFixed);
    }
    return newFn;
}
```

you cannot write this here;

```
Function.prototype.bind = function (objContext, ...fixArgs) {
    const oldFn = this;
    const newFn = function (...nonFixedArgs) {
        return oldFn.call(objContext, ...fixArgs, ...nonFixedArgs);
    }
    return newFn;
}
```

oldFn → normal Fn

* const bindedFn = normal.bind(obj, 1);

→ function (...nonfixed) {
return normalFn.call(obj, 1, ...nonFixed);

}

th class > JS bind.js > ...

```
1  Function.prototype.bind = function (objContext, ...fixArgs) {
2      const oldFn = this;
3      const newFn = function (...nonFixedArgs) {
4          return oldFn.call(objContext, ...fixArgs, ...nonFixedArgs);
5      }
6      return newFn;
7  }
8
```

const bindedFn = normal.bind(obj, 1);

↳ function (...arg) {

return normalFn (obj, 1, ...arg);

}

oldFn

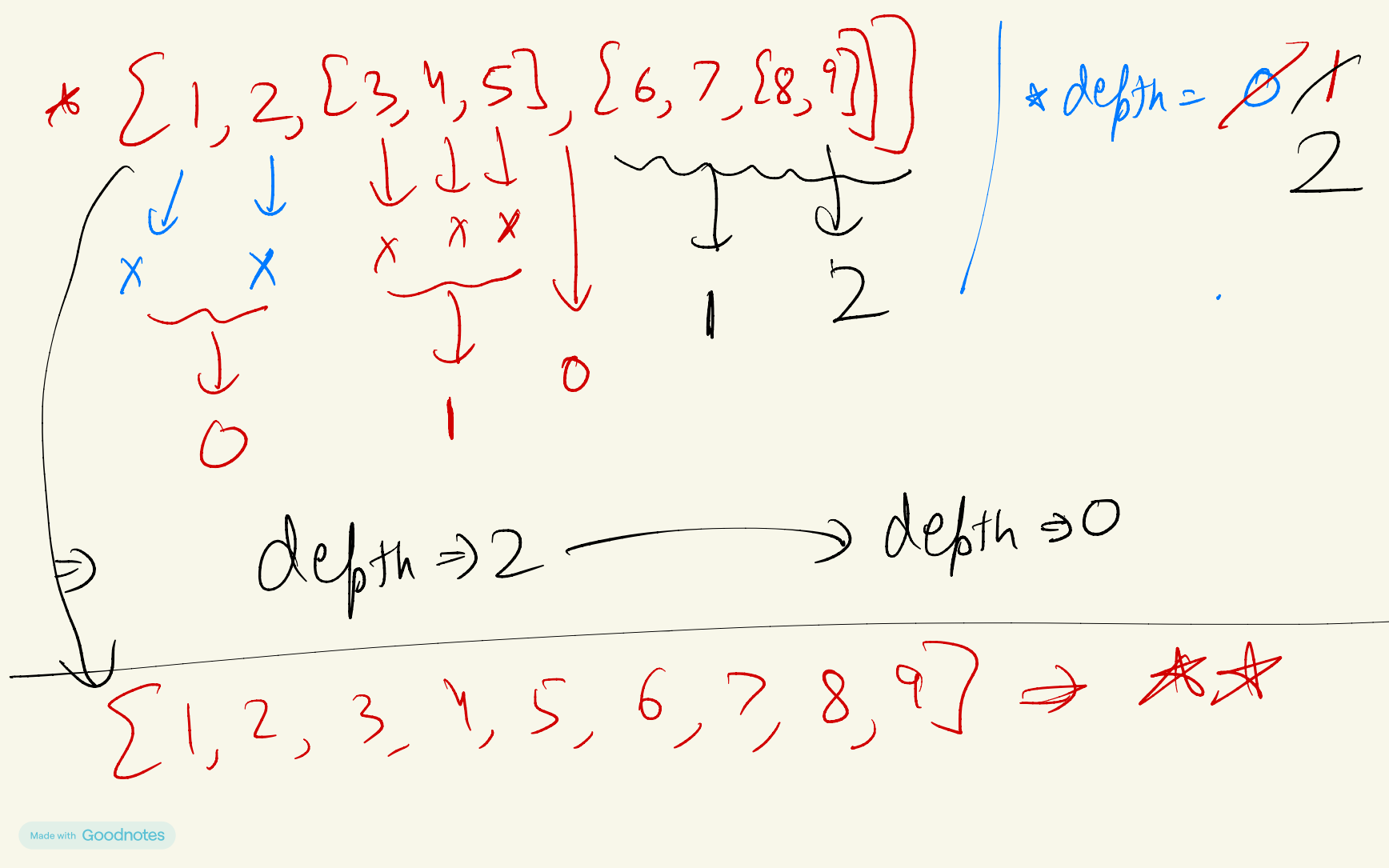
↓

normal Fn

[1, 2, 3, [4,5]]

* arr[3][0] ⇒ depth of 1.

[1, 2, 3, 4, 5] ⇒ depth of 0



[1, 2, [3, 4, 5], [6, 7, [8, 9]]]

```
function myFlat (arr){
    cons res = [];
    arr.forEach (item => {
        if (Array.isArray (item)){
            const miniAns = myFlat (item);
            result.push (...miniAns);
        }
        else res.push (item);
    }
    return res;
}
```

```
12  function myFlat(arr) {
13      const res = [];
14      arr.forEach(element => {
15          if (Array.isArray(element)) {
16              const miniAns = myFlat(element);
17              res.push(...miniAns);
18          }
19          else {
20              res.push(element);
21          }
22      })
23      return res;
24  }
```

[1, 2, [3,4,5], [6,7,[8,9]]]

(2)

* myFlat([6,7,[8,9]])

[6, 7, 8, 9]

(1)

* res

[1, 2, 3, 4, 5, 6, 7, 8, 9]