First class Function

Not

(i) If function in language are treated like other variables.

* So function assigned to any other variable or object

pass as an argument (callback)

or

return by another function.

* Is treat function as first class citizen to Function (simply value) just another type of object.

Eg: Pass as argument

```
function sayHello() {
    return 'Hello';
}
function mainFunction(argFn, name) {
    console.log(argFn(), name);
}
mainFunction(sayHello, 'John');
```

o/p

Hello John

Concatenate

MAY 2020

Su	Mo	Tu	We	Th	Fr	Sa
31					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

→ Can Be returned by another function

Eg:
```
function add() {
    return function() {
        console.log("added");
    }
}
```

→ Can be assigned as a value to a { variable / object / array }

1	Variable	`const abc = function() {}`
2	Object	`const newObj = { getName: function() { } }`
3	Array	`let arr = []; arr.push(function() {})`

→ Functional Programming → We pass functions as parameters values ← as to other functions, return them also

→ Think Code } → in terms of → Functions.

→ Higher Order Function (HOF) ← only possible because of First class Func

→ Functions → operate → on other → by → taking them as argument or By Returning them

→ Function → receives → function as an argument or → returns → function as O/P.

→ Eg: sort(), filter(), map(), reduce().

130

MAY 2020

1) map()

Week 21 | 143-223

22 Friday

~ Create new array by calling callback function provided as an argument on every element in i/p array.

~ Take every returned value from → CallBack Function → create new array by using those values

~ CallBack function ~ pass → map() ~ accepts ~ 3 Arguments → Element, index, array

Without HoF

```
<script>
  Const arr1 = [1,2,4,6,8];
  Const arr2 = [];
for(let i=0; i<arr1.length; i++){
  arr2.push(arr1[i] *2);
}
Console.log(arr2);
</script>
```

o/p : Array(5) [2,4,8,12,16]

i | i<arr1.length | arr2.push(i)

0	0<5	✓	1x2=2
1	1<5	✓	2x2=4
2	2<5	✓	4x2=8
3	3<5	✓	6x2=12
4	4<5	✓	8x2=16
5	5<5	x	—

With HoF

```
<script>
  const arr1 = [1,2,4,6,8]
  Const arr2 = arr1.map(function(item){
    return item*2;
  });
  Console.log(arr2);
</script>
```

o/p : (5) [2,4,8,12,16]

```
Const arr2 = arr1.map(item => item*2);
```

by CallBack function provided Pass test that

Create New array → with → all elements

2) filter()

~ accepts → 3 Arguments

eg:-

```
Const persons = [{name:'m', age:15}, {name:'n', age:16}];
Const filterage = persons.filter(person => person.age > 15);
Console.log(filterage);
```

MAY 2020

Su	Mo	Tu	We	Th	Fr	Sa
31					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

B1

Single o/p value (in results) — Calling array

3) reduce()

- Executes CallBack function on each member of the Calling array
- Accepts 2 Parameters
  - Reducer function (CallBack) → 4 Parameters (accept)
    - current index
    - accumulator
    - current value
    - Source Array
  - Optional - Initial value
- If Initial value provided then
  - accumulator = initial value
  - Current value = 1st element in array
- not provided then
  - Accumulator = 1st element in array
  - Current Value = 2nd element in array

eg) Every time reducer function called on each value in array — keeps accumulator — result of previous operation — from returned

Current value — set current value of array.

```
<script>
const pes = [1,2,3,4,5];
const sum = pes.reduce(function(acc, current){
    return acc+current;
});
console.log(sum);
</script>
```

o/p: 15

acc = 1 3 6 10 current = 2 3 4 5 ... = 15

a = 1
c = 2

	a	c
0	1	2
1	3	3
2	6	4
3	10	5

132

```
const sum = pers.reduce(function(acc, current){ return acc + current.age }, 0);
```

Eg2) Use this line & modify → o/p : 20

4) Create Own HoF → Exp → mapForEach (accept [array → arrstr, callBack function → item, length])

["Java", "C", "Js"]
0 1 2

newarr = [ ]

i	i<arr.length	i++
0	0<2	1

fn(arr[0]) fn("Java")

Eg:

```
<script>
const arrstr = ["Java", "C", "Js"];
function mapForEach(arr, fn){
    const newarr = [];
    for(let i=0; i<arr.length; i++){
        newarr.push(fn(arr[i]));
    }
    return newarr;
}
const lenarr = mapForEach(arrstr, function(item){
    return item.length;
});
console.log(lenarr);
</script>
```

item = arr[i]

[4,]

o/p : 4,1,2

Reusability ← Reduce code size ← HoF

First order	HoF
do → functions	do → function in mathematical sense
They → treat → variables	Function → Return function
also → passed → into → arguments → other functions	Take one or more (functions) as argument → accept parameter
behave like variable → passed as argument → to HoF	→ map() → some() → filter() → setTimeout() → reduce()

MAY 2020

Su	Mo	Tu	We	Th	Fr	Sa
31						
3	4	5	6	7	1	2
10	11	12	13	14	8	9
17	18	19	20	21	15	16
24	25	26	27	28	29	30

Eg:- function sayHello() {
    return 'Hi';
}

function hof(sayHelloCallBack, name) {
    console.log(sayHelloCallBack(), name);
}

hof(sayHello, 'John');

Higher Order Function → Main Function

CallBack Function

Eg:-

```
<body>
    <p id="p1">Change</p>
    <button onclick="redfont()">Red</button>
    <button onclick="bluefont()">Blue</button>
<script>
    let ida = document.getElementById("p1");
    const redfont = changeColor("red");
    const bluefont = changeColor("blue");
    function changeColor(color) {
        return function() {
            ida.style.color = color;
        }
    }
</script>
</body>
```

5) Some() → Check if any array elements pass test (provide as callback function) → execute callback function once for each array element

Returns true → if function returns true for one of array elements

false → if function returns false for all of array elements

B4 Eg:

Syntax:- array.someFunction(value, index, arr) this)

```
<script>
    let checkMin = (age) => age > 18;
    const arrage = [12, 13, 17, 34, 23, 33];
    let check = arrage.some(checkMin);
    console.log(check); //true
</script>
```

18 ✓ 12 < 3 ✗

Node.js

true

o/p

Map()

The **map()** method creates a **new array** by calling a provided function on each element in the original array.

It doesn't change the **original array**; instead, it **returns a new array** with the results of the function applied to each element.

```
// Example: Doubling each number in an array

const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = numbers.map(num => num * 2);
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
```

Filter()

The **filter()** method creates a **new array** with all elements that pass the test implemented by the provided function.

Like **map()** It doesn't change the **original array**; instead, it returns a new array with only the elements that meet the condition.

```
// Example: Filtering even numbers from an array

const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 ===
0);
console.log(evenNumbers); // Output: [2, 4]
```

Reduce()

The **reduce()** method applies a function against an **accumulator** and each element in the array to reduce it to a single value.

It takes a callback function with an accumulator and current value as arguments and **returns a single value**.

```
// Example: Summing all numbers in an array

const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue)
=> accumulator + currentValue, 0);
console.log(sum); // Output: 15 (1 + 2 + 3 + 4 + 5)
```

Summary

Use **map()** when you want to transform each element of an array into something else and get a **new array** with those transformed elements.

Use **filter()** when you want to get a new array with only the elements that meet **certain criteria.**

Use **reduce()** when you want to transform an array into a **single value**, like summing up all the elements or finding the maximum value.