

HOFs & Callbacks

Basics of Functions:

Function Statement:

- The function statement declares a function. A declared function is "saved for later use", and will be executed later, when it is invoked (called).

```
function javascript(){
    console.log("Welcome to JS")
}

javascript()
```

Function expression:

- functions are like heart to JavaScript, beautiful feature of a function is that you can assign it to a variable.

```
var b = function(){
    console.log("Welcome to JS")
}

b()
```

Anonymous function :

- A function without a name is called anonymous function
- Anonymous functions are used as values , i.e. you can use it to assign it to some variables. In the above snippet the function which we assign to variable b is an anonymous function

Difference Between Parameters and Arguments :

- *Parameters* are variables listed as a part of the function definition.
- *Arguments* are values passed to the function when it is invoked.

```
function sum( a, b, c ) {}; // a, b, and c are the parameters
sum( 1, 2, 3 ); // 1, 2, and 3 are the arguments
```

Think of vaccination scenario



- Many People are queued up and as a doctor what would you tell your staff to do:
 1. Everyone has a token of their number in the line. You go with a token 0 and match it with the first person and vaccinate him, then you scratch the 0 on your token and make it 1, then go to next person and vaccinate, and so on.
 2. Go and vaccinate everyone in the line one by one.
- Lets take array of persons

```
var persons = ['Chandra', 'Varun', 'Nrupul', 'Prateek', 'Ama
n'];
```

- Let's write a function for vaccination

```
function vaccinate(person) {
  console.log(person + 'has been vaccinated.')
}
```

- Instead of going to each and every person, lets use for loop

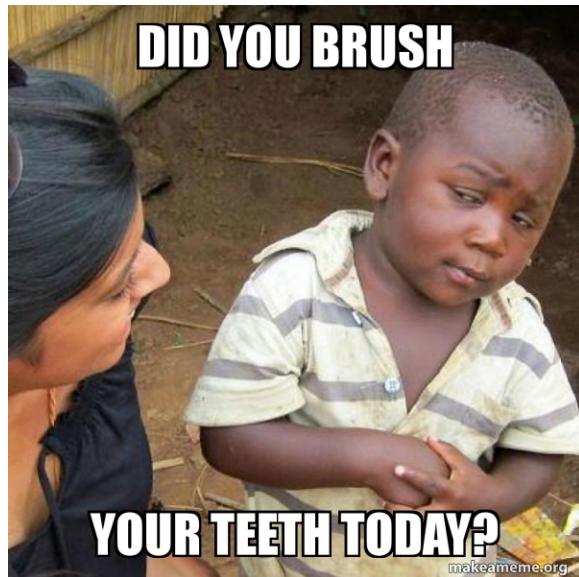
```
for (var i = 0; i < persons.length; i++) {
  vaccinate(persons[i]);
}
```

- You can also use `map` function instead of `for` loop

```
persons.map(vaccinate)
```

- **The surprising thing that happened is we passed a function name as an argument!**
- What is `vaccinate` here ? it's a callback function

Callback functions:



- let's write a function for eatBreakfast

```
function eatBreakfast(item){  
    console.log("I will eat"+" "+item +" "+ "as my breakfas  
t")  
}  
  
eatBreakfast("idly")
```

Output : I will eat idly as my breakfast

- Now let's pass number as argument along with string

```
function eatBreakfast(item,time){  
    console.log("I will eat"+" "+item +" "+ "as my breakfas  
t"+ "at" + " "+ time)  
}
```

```
eatBreakfast("idly", 9)
```

Output : I will eat idly as my breakfast at 9

- Now let's try to pass functions as argument along with strings and numbers

```
function eatBreakfast(item, time){  
  
    console.log("I will eat" + " " + item + " " + "as my breakfas  
t" + "at" + " " + time)  
}  
  
  
function doBrush(){  
    console.log("First brush your teeth")  
}  
  
eatBreakfast("idly", 9, doBrush)
```

Output : I will eat idly as my breakfast at 9

- We have passed function as argument but how to access callback function

```
function eatBreakfast(item, time, doBrush){  
    doBrush()  
    console.log("I will eat" + " " + item + " " + "as my breakfas  
t" + "at" + " " + time)  
}  
  
  
function doBrush(){  
    console.log("First brush your teeth")
```

```
}
```

```
eatBreakfast("idly", 9, doBrush)
```

Output :

First brush your teeth

I will eat idly as my breakfast at 9

HOF - Sweets Analogy

- Suppose you go to a sweet shop to buy some sweets



- Sweetshop has so many variety of sweets



- You will be doing one kind among these
 - One sweet from all varieties available for eg: 1 kova, 1 laddu, 1 gulabjamun, etc.



www.shutterstock.com · 1337715272

- Only one kind of sweet eg: kova



- Mixing all kinds of sweets in shop itself and your stomach will be like this 😊



forEach:

Instructor Task:

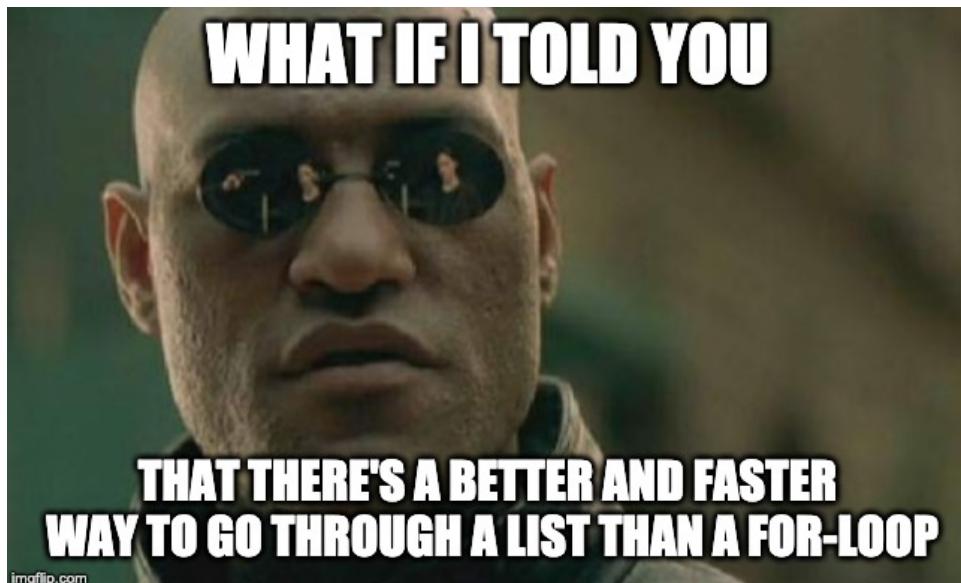
- Lets take example of this sweets menu

```
var sweets = ["kova", "gulabjamun", "laddu", "mysorepak", "badsha
w"]
```

- As a foodie, I wanted all of these items in my plate, so I will go to each and every dish and pick it up, and also I can use for loop for this

```
var sweets = ["kova", "gulabjamun", "laddu", "mysorepak", "badsha
w"]

for (var i = 0; i < sweets.length; i++){
  console.log(food_menu[i])
}
```



- Syntax of `forEach`

```
var sweets = ["kova", "gulabjamun", "laddu", "mysorepak", "badsha
w"]
```

```
sweets.forEach(function (elem, index) {  
    console.log(elem)  
})
```

Output :

```
kova  
gulabjamun  
laddu  
mysorepak  
badshaw
```

- here elem is each sweet individually
- index is index number.

Warning :

- forEach has extra charges



- To pack those sweets in a box, we need to pay extra charges



```
var sweets = ["kova", "gulabjamun", "laddu", "mysorepak", "badshaw"]
var box = []
sweets.forEach(function (elem, index) {
  box.push(elem)
})

console.log(box) //['kova', 'gulabjamun', 'laddu', 'mysorepak', 'badshaw']
```

- Here creating extra array is extra charge.

map:

- map is similar to forEach, only difference is map doesn't have any additional charges



```
var sweets = ["kova", "gulabjamun", "laddu", "mysorepak", "badshaw"]
var output= sweets.map(function (elem, index) {
    return elem
})

console.log(output)// [ 'kova', 'gulabjamun', 'laddu', 'mysor
epak', 'badshaw' ]
```

- map method will return you a box(array) along with sweets

Difference between forEach and map

forEach



map



forEach	map
Return value: <code>undefined</code>	Return value: newArray will be created
Original Array: not modified	Original Array: not modified
forEach method is Not chainable	map method is chainable

filter:



- If you want only one sweet for eg:kova, then we will use filter
- Filter also doesn't have any additional charges
- It will return you a box(array) along with sweets

```
var sweets = ["kova", "gulabjamun", "laddu", "mysorepak", "badshaw"]
var output = sweets.filter(function (elem, index) {
    return elem === "kova"
})

console.log(output) // ["kova"]
```

reduce:

- The `.reduce()` method iterates through an array and returns a single value.
- There are two scenarios
 - Without initial value

- With initial value
- **How reduce() works without an initial value**
- The code below shows what happens if we call `reduce()` with an array and no initial value.

```
const array = [15, 16, 17, 18, 19];

array.reduce(function (acc, el) {
  return acc+el;
});
```

callback iteration	acc	current value(el)	acc+el (stores in acc)
first call	15	16	31
second call	31	17	48
third call	48	18	66
fourth call	66	19	85

- **How reduce() works with an initial value**
- Here we reduce the same array using the same algorithm, but with an *initialValue* of `10` passed as the second argument to `reduce()`

```
let array = [10, 16, 17, 18, 19];

let addNums=function (acc, cv) {
  return acc+cv;
}

array.reduce(addNums, 10);
```

callback iteration	acc	current value(cv)	acc+cv (stores in acc)
first call	10	15	25
second call	25	16	41
third call	41	17	58
fourth call	58	18	76
fifth call	76	19	95

Sorting:

the `sort()` method is used to sort the elements of an array in place and returns the sorted array. The default sort order is built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.

Here is an example of using the `sort()` method to sort an array of numbers in ascending order:

Copy code

```
let numbers = [4, 2, 9, 1, 3, 6];
numbers.sort();
console.log(numbers); // [1, 2, 3, 4, 6, 9]
```

You can also pass a compare function as an argument to the `sort()` method to define a custom sort order. The compare function should take two arguments (a and b) and return a negative, zero, or positive value, depending on the arguments, like:

- a negative value if a should be sorted lower than b,
- a positive value if a should be sorted higher than b,
- 0 if a and b are equal.

Here is an example of using a compare function to sort an array of objects by the value of their "age" property:

```

let shapes = [
  {sides: 4, color: 'red'},
  {sides: 3, color: 'blue'},
  {sides: 4, color: 'blue'},
  {sides: 3, color: 'red'}
];

// Output: [
//   {sides: 3, color: 'blue'},
//   {sides: 4, color: 'blue'},
//   {sides: 3, color: 'red'},
//   {sides: 4, color: 'red'}
// ]

```

Chaining

Till now, we've seen a few higher order functions which give us power to apply a logic using less lines of code, right?

What if I told you that is not all that JS provides us. We can even combine these higher order functions. Let's solve a few problems and learn the concept of chaining through the problems.

method	Input	return value
forEach	array	undefined
map	array	array
filter	array	array
reduce	array	single value

```
forEach.map  
map.filter  
filter.reduce  
forEach.filter  
forEach.reduce  
map.reduce
```