Higher Order Functions

A function that accepts and/or returns another function is called

a higher-order function. It is higher-order because instead of strings, numbers, or booleans, it goes higher to operate on functions.

Array Callback Methods

1. forEach
2. map
3. filter
4. find
5. findIndex
6. reduce
7. some
8. every
9. sort
10. flatMap

forEach : Anonymously Callback var nums = [ 1, 2, 3, 4, 5 ]; nums.forEach(function(num)

{

console.log(`Square of ${num} is ${num\*num}`);

});

forEach : Named Callback

var nums = [ 1, 2, 3, 4, 5 ]; function square(num)

{

console.log(`Square of ${num} is ${num\*num}`);

}

nums.forEach(square);

forEach : Named Callback Arrow Function const nums = [ 1, 2, 3, 4, 5 ];

const square = (num)=>

{

console.log(`Square of ${num} is ${num\*num}`);

}

nums.forEach(square);

forEach : Anonymously Callback Arrow Function const nums = [ 1, 2, 3, 4, 5 ]; nums.forEach((num)=>

{

console.log(`Square of ${num} is ${num\*num}`);

});

Map

creates a new array populated with the results of calling a provided function on every element in the calling array.

map: Named Callback

const nums = [ 1, 2, 3, 4, 5 ]; function square(num)

{

return num \* num;

}

const square\_nums = nums.map(square); console.log(square\_nums);

map: Anonymously Callback

var nums = [ 1, 2, 3, 4, 5 ];

const square\_nums = nums.map(function(num)

{

return num \* num ;

});

console.log(square\_nums);

map: Named Callback Arrow Function const nums = [ 1, 2, 3, 4, 5 ]; const square = (num)=>

{

return num \* num;

}

const square\_nums = nums.map(square); console.log(square\_nums);

map: Anonymously Callback Arrow Function const nums = [ 1, 2, 3, 4, 5 ] ;

const square\_nums = nums.map((num)=>

{

return num \* num;

});

console.log(square\_nums);

map : Named Callback Arrow Function Implicit Return const nums = [ 1, 2, 3, 4, 5 ];

const square = (num)=> num \* num ; const square\_nums = nums.map(square); console.log(square\_nums);

map : Anonymously Callback Arrow Function const nums = [ 1, 2, 3, 4, 5 ];

const square\_nums = nums.map((num)=> num \* num) ; console.log(square\_nums);

find

returns the value of first element in the array that satisfies the provided testing condition

const ages = [7, 18, 23, 15, 25];

const adult = ages.find((age)=> 18 <= age); console.log(adult); // 18

findIndex

returns the index of first element in the array that satisfies the provided testing condition

const ages = [7, 18, 23, 15, 25];

const adult = ages.findIndex((age)=> 18 <= age ); console.log(adult);

filter

creates a new array with all elements that pass the test implemented by the provided function.

const ages = [7, 18, 23, 15, 25];

const adults = ages.findIndex((age)=> 18 <= age ); console.log(adults);

some

returns true if any of the array elements pass the test function. const ages = [7, 18, 23, 15, 25];

const isAdultPresent = ages.some((age)=> 18 <= age ); console.log(isAdultPresent);

every

returns true if all elements of the array pass the test function. const ages = [7, 18, 23, 15, 25];

const isAdultPresent = ages.every((age)=> 18 <= age ); console.log(isAdultPresent);

sort

The default sort is very weird. It converts all the array elements as strings and sort them as strings which leads to very odd behavior. const prices = [400.50, 3000, 99.99, 35.99, 12.00, 9500];

prices.sort();

console.log(prices) // [12, 3000, 35.99, 400.5, 9500, 99.99]

Ascending Order

const prices = [400.50, 3000, 99.99, 35.99, 12.00, 9500];

prices.sort((a,b)=> a-b); console.log(prices)

Descending Order

const prices = [400.50, 3000, 99.99, 35.99, 12.00, 9500];

prices.sort((a,b)=> b-a); console.log(prices)

Descending Order But Different New Array

const prices = [400.50, 3000, 99.99, 35.99, 12.00, 9500];

const sorted\_prices = prices.slice().sort((a,b)=> a-b); console.log(sorted\_prices);

Sorting data with their ID’s

const data = [

{

id: 23, name: 'John',

},

{

id: 1,

name: 'Peter',

},

{

id: 111,

name: 'Thomas',

},

{

id: 97,

name: 'Jessica',

},

{

id: 8,

name: 'Jordan',

}

]

data.sort((obj1, obj2)=>

{

return obj1.id - obj2.id;

})

console.log(data);

/\* [

{ id: 1, name: 'Peter' },

{ id: 8, name: 'Jordan' },

{ id: 23, name: 'John' },

{ id: 97, name: 'Jessica' },

{ id: 111, name: 'Thomas' }

]

\*/

Sorting data with their names var data = [

{ name: 'Edward', value: 21 },

{ name: 'Sharpe', value: 37 },

{ name: 'Andrew', value: 45 },

{ name: 'Thomas', value: -12 },

{ name: 'Magneto', value: 13 },

{ name: 'Zenta', value: 37 }

];

data.sort((a, b) =>

{

var nameA = a.name.toUpperCase( ); // ignore upper and lowercase var nameB = b.name.toUpperCase(); // ignore upper and lowercase if (nameA < nameB)

{

return -1;

}

if (nameA > nameB)

{

return 1;

}

// names must be equal return 0;

});

console.log(data);

/\*

[

{ name: 'Andrew', value: 45 },

{ name: 'Edward', value: 21 },

{ name: 'Magneto', value: 13 },

{ name: 'Sharpe', value: 37 },

{ name: 'Thomas', value: -12 },

{ name: 'Zenta', value: 37 }

]

\*/

reduce

It executes the reducer function on each element of the array, resulting in a single value.

const nums = [ 3, 5, 7, 6 ]

var result = nums.reduce((accumulator, currentvalue)=> accumulator + currentvalue);

console.log(result); //21

|  |  |  |  |
| --- | --- | --- | --- |
| Callback | Accumulator | Current Value | Return Value |
| First Call | 3 | 5 | 8 |
| Second Call | 8 | 7 | 15 |
| Third Call | 15 | 6 | 21 |

const grades = [ 87, 64, 96, 92, 88, 99, 73, 70, 64 ]

var result = grades.reduce((accumulator, currentvalue)=> Math.max(accumulator, currentvalue)); console.log(result);

Similarly, we can use Math.min() function. flatMap

The flatMap() method returns a new array formed by applying a given callback function to each element of the array, and then flattening the result by one level. It is identical to a [map()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) followed by a [flat()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/flat) of depth 1 (arr.map(...args).flat()), but slightly more efficient than calling those two methods separately.

const pairs = [ [2, 6], [8, 2], [5, 9] ] const arr = pairs.flatMap((pair)=>

{

return [ pair[0] + pair[1]];

})

console.log(arr); // [ 8, 10, 14 ]