Map

The `map()` method is used for creating a new array from an existing one, applying a function to each one of the elements of the first array.

Syntax

```
var new_array = arr.map(function callback(element, index, array) {
    // Return value for new_array
}[, thisArg])
```

In the callback, only the array `element` is required. Usually some action is performed on the value and then a new value is returned.

Example

In the following example, each number in an array is doubled.

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(item => item * 2);
console.log(doubled); // [2, 4, 6, 8]
```

Filter

The `filter()` method takes each element in an array and it applies a conditional statement against it. If this conditional returns true, the element gets pushed to the output array. If the condition returns false, the element does not get pushed to the output array.

Syntax

```
var new_array = arr.filter(function callback(element, index, array) {
    // Return true or false
}[, thisArg])
```

The syntax for `filter` is similar to `map`, except the callback function should return `true` to keep the element, or `false` otherwise. In the callback, only the `element` is required.

Examples

In the following example, odd numbers are "filtered" out, leaving only even numbers.

```
const numbers = [1, 2, 3, 4];
const evens = numbers.filter(item => item % 2 === 0);
console.log(evens); // [2, 4]
```

In the next example, `filter()` is used to get all the students whose grades are greater than or equal to 90.

```
const students = [
  { name: 'Quincy', grade: 96 },
  { name: 'Jason', grade: 84 },
  { name: 'Alexis', grade: 100 },
  { name: 'Sam', grade: 65 },
  { name: 'Katie', grade: 90 }
];

const studentGrades = students.filter(student => student.grade >= 90);
return studentGrades; // [ { name: 'Quincy', grade: 96 }, { name: 'Alexis', grade: 100 }, { name: 'Katie', grade: 90 } ]
```

Reduce

The `reduce()` method reduces an array of values down to just one value. To get the output value, it runs a reducer function on each element of the array.

Syntax

```
arr.reduce(callback[, initialValue])
```

The `callback` argument is a function that will be called once for every item in the array. This function takes four arguments, but often only the first two are used.

- *accumulator* - the returned value of the previous iteration
- *currentValue* - the current item in the array
- *index* - the index of the current item
- *array* - the original array on which reduce was called
- The `initialValue` argument is optional. If provided, it will be used as the initial accumulator value in the first call to the callback function.

Examples

The following example adds every number together in an array of numbers.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce(function (result, item) {
  return result + item;
}, 0);
console.log(sum); // 10
```

In the next example, `reduce()` is used to transform an array of strings into a single object that shows how many times each string appears in the array. Notice this call to reduce passes an empty object `{}` as the `initialValue` parameter. This will be used as the initial value of the accumulator (the first argument) passed to the callback function.

```
var pets = ['dog', 'chicken', 'cat', 'dog', 'chicken', 'chicken', 'rabbit'];

var petCounts = pets.reduce(function(obj, pet){
    if (!obj[pet]) {
        obj[pet] = 1;
    } else {
        obj[pet]++;
    }
    return obj;
}, {});

console.log(petCounts);

/*
Output:
 {
  dog: 2,
  chicken: 3,
  cat: 1,
  rabbit: 1
 }
 */
```

FOR EACH : The forEach() method calls a function for each element in an array. The forEach() method is not executed for empty elements.

Syntax array.forEach(function(currentValue, index, arr), thisValue)

EX:- let sum = 0;

```
const numbers = [65, 44, 12, 4];
numbers.forEach(myFunction);

function myFunction(item) {

sum += item; }
```