

Array Collection Methods

We look at a number of methods of array objects that capture the idea of a “collection of values”, and offer ways to manipulate the entire collection, rather than performing an iterative for loop.

These are often called “higher-order functions”.

The idea is simple: We forget about the fact that an array is in fact an array, which a specific length and value at each index, and we treat it instead as a *collection* of elements. We might then want to perform some standard operations on these elements:

- Perform an **action** for each element (forEach)
- **Transform** each element into a new element based on some specific function (map)
- **Restrict** the list of elements to only those that match certain criteria (filter)
- **Accumulate** some value as we traverse the elements, and produce one final result (reduce)

If you think through most of the cases where you traversed the elements of an array, you were likely performing some of these steps.

Relevant links

- Flanagan’s Book, section 7.9
- Array object documentation¹

Notes

Here is a summary of array higher-order methods:

forEach² Apply a function to each element of the array

map³ Apply a function to each element of the array and form a new array from the results

reduce⁴ Apply a function on an accumulator and each array value in order, resulting in one final reduced result.

filter⁵ Given a predicate, return a new array with those items from the array that evaluate to true.

every⁶ Given a predicate, return true or false depending on whether all items from the array satisfy it or not.

some⁷ Given a predicate, return true or false depending on whether at least one item from the array satisfies it or not.

We will discuss each method in turn.

¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

forEach

The `forEach` method is really our bread and butter for performing something on each element of the array. It takes as argument a function `f`, and calls that function once for every element of the array, passing it as arguments the value, the index, and the whole array object. These functions are often called “callbacks” in the documentation.

NOTE: This method will skip indices that are undefined (but not indices whose value is “undefined”).

A simple example would be a log function:

```
let a = [4,3,1,5];
a.forEach((v, i) => console.log("Found the value " + v + " at index " + i));
```

map

The `map` method returns a new array from the results of applying the given function `f` to the original array. It will also skip undefined indices, preserving them in the result. Here is an example where we increment each entry by its index. It is important to note that the original array remains unaffected.

```
a.map((v, i) => v + i);
a;
```

reduce

`reduce` is the most powerful of these methods, and the harder to understand. Essentially it starts with an initial value, and accumulates into it the results of applying a certain function to the values in the array, along with the accumulated values. An example will probably be better. Consider the following code:

```
a.reduce((acc, v) => acc + v, 10);
```

So what this code will do is start with an initial value of 10 for `acc`. It will then apply the function to 10 and the first value in the array, 4, resulting in 14. It will then proceed to apply the function (addition) to this 14 and the next value in the array, namely 3, resulting in 17. Then it will add this 17 to 1, resulting in 18, and finally it will add this 18 to 5, producing the final result of 23. It will then return that value.

So `a.reduce(f, init)` is roughly equivalent to the following code:

```
let acc = init;
for (let i = 0; i < a.length; i += 1) {
  acc = f(acc, a[i]);
}
return acc;
```

In reality, the function `f` receives extra parameters, namely the index `i` and the whole array.

One of the reasons to use these functions is that they can be better optimized by the Javascript interpreter/compiler, and thus typically run faster. The following timing test will attest to that.

```
let A = []
for (let i = 0; i < 100000; i += 1) { A.push(Math.random()); }
let times = [];
for (let j = 0; j < 10; j += 1) {
  let t = new Date();
  A.reduce((a, b) => a + b, 0);
  times.push(new Date() - t);
}
console.log(times)
times = [];
for (let j = 0; j < 10; j += 1) {
  let t = new Date();
  let s = 0;
  for (let i = 0; i < A.length; i += 1) { s += A[i]; }
  times.push(new Date() - t);
}
console.log(times);
```

filter

The filter method expects a predicate, namely a function that returns a boolean. It will call this function for each value, and will add to a new array those values that return true from the predicate.

As usual with the other methods here, the predicate will be given two more arguments, the index and the whole array.

As an example, the following will retrieve the even-indexed elements in the array:

```
a.filter((v, i) => i % 2 === 0);
```

every and some

These methods take the same argument as filter, and apply it to the array values until they can resolve their result.

every will return false the moment it encounters a value that the predicate evaluates to false, and otherwise (if it makes it to the end of the array) it will return true.

some is sort of the opposite. It will return true the moment it finds a value that the predicate evaluates to true, and otherwise (if it makes it to the end of the array) it will return false.

Makes sure you understand how these two methods will behave on an empty array.

Practice

1. Using `every`, write a call that will tell us if all numbers in an array of numbers are even.
2. Using `filter`, keep from an array of strings only those with length no more than 20.
3. Using `reduce`, compute given an array of numbers the sum of squares of those numbers.
4. Using `map`, given an array of strings produce an array of the corresponding lengths of those strings.
5. Using a combination of `filter` and `map`, produce from an array of numbers a corresponding array of the squares of only those numbers that are positive. You can “chain” the two calls like so:

```
arr.filter(function(v) { ... })  
  .map(function(v) { ... });
```