Array iteration methods summarized

Array iteration methods summarized

Array Iteration

https://gist.github.com/ljharb/58faf1cfcb4e6808f74aae4ef7944cff

While attempting to explain JavaScript's `reduce` method on arrays, conceptually, I came up with the following - hopefully it's helpful; happy to tweak it if anyone has suggestions.

Intro

JavaScript Arrays have lots of built in methods on their prototype. Some of them *mutate* - ie, they change the underlying array in-place. Luckily, most of them do not - they instead return an entirely distinct array. Since arrays are conceptually a contiguous list of items, it helps code clarity and maintainability a lot to be able to operate on them in a "functional" way. (I'll also insist on referring to an array as a "list" - although in some languages, `List` is a native data type, in JS and this post, I'm referring to the concept. Everywhere I use the word "list" you can assume I'm talking about a JS Array) This means, to perform a single operation on the list as a whole ("atomically"), and to return a *new* list - thus making it much simpler to think about both the old list and the new one, what they contain, and what happened during the operation.

Below are some of the methods that *iterate* - in other words, that operate on the entire list, one item at a time. When you call them, you provide a *callback function* - a single function that expects to operate on one item at a time. Based on the Array method you've chosen, the callback gets specific arguments, and may be expected to return a certain kind of value - and (except for `forEach`) the return value determines the final return value of the overarching array operation. Although most of the methods are guaranteed to execute for *each* item in the array - for all of them - some of the methods can stop iterating partway through; when applicable, this is indicated below.

All array methods iterate in what is traditionally called "left to right" - more accurately (and less ethnocentrically) from index `0`, to index `length - 1` - also called "start" to "end". `reduceRight` is an exception in that it

iterates in reverse - from `end` to `start`.

Special Mention: `Array.from`

`Array.from`, introduced in ES6/ES2015, accepts any array, "arraylike", or "iterable". An "arraylike" object is anything with a `length`, which includes arrays, strings, and DOM NodeLists (as produced by `document.querySelectorAll`, eg). An "iterable" is anything that has a `Symbol.iterator` method, which includes arrays, strings, NodeLists, Maps, Sets, and potentially many more.

To efficiently convert any non-array into one for the purpose of using the below methods, you can utilize `Array.from`'s "mapper" argument:

```
const tagNames = Array.from(document.querySelectorAll('.someClass'),
(el) => el.tagName);

const codePointValues = Array.from('some string with 💩 emoji 💩!',
(codePoint) => codePoint.codePointAt(0));

const mapValues = Array.from(map, ([key, value]) => value);
```

`forEach`:

- *callback answers*: here's an item. do something nutty with it, i don't care what.
- *callback gets these arguments*: `item`, `index`, `list`
- *final return value*: nothing - in other words, `undefined`
- *example use case*:

```
[1, 2, 3].forEach(function (item, index) {
  console.log(item, index);
});
```

`map`:

- *callback answers*: here's an item. what should i put in the new list in its place?
- *callback gets these arguments*: `item`, `index`, `list`
- *final return value*: list of new items
- *example use case*:

```
const three = [1, 2, 3];
const doubled = three.map(function (item) {
  return item * 2;
});
console.log(three === doubled, doubled); // false, [2, 4, 6]
```

`filter`:

- *callback is a predicate* - it should return a truthy or falsy value
- *callback answers*: should i keep this item?
- *callback gets these arguments*: `item`, `index`, `list`
- *final return value*: list of kept items
- *example use case*:

```
const ints = [1, 2, 3];
const evens = ints.filter(function (item) {
  return item % 2 === 0;
});
console.log(ints === evens, evens); // false, [2]
```

`reduce`:

- *callback answers*: here's the result from the previous iteration. what should i pass to the next iteration?
- *callback gets these arguments*: `result`, `item`, `index`, `list`
- *final return value*: result of last iteration
- *example use case*:

```
// NOTE: `reduce` and `reduceRight` take an optional "initialValue"
argument, after the reducer callback.
// if omitted, it will default to the first item.
const sum = [1, 2, 3].reduce(function (result, item) {
  return result + item;
}, 0); // if the `0` is omitted, `1` will be the first `result`, and
`2` will be the first `item`
```

`reduceRight`: (same as `reduce`, but in reversed order: last-to-first)

`some`:

- *callback is a predicate* - it should return a truthy or falsy value
- *callback answers*: does this item meet your criteria?
- *callback gets these arguments*: `item`, `index`, `list`
- *final return value*: `true` after the first item that meets your criteria, else `false`
- **note**: stops iterating once it receives a truthy value from your callback.
- *example use case*:

```
const hasNegativeNumbers = [1, 2, 3, -1, 4].some(function (item) {
  return item < 0;
});
console.log(hasNegativeNumbers); // true
```

`every`:

- *callback is a predicate* - it should return a truthy or falsy value
- *callback answers*: does this item meet your criteria?
- *callback gets these arguments*: `item`, `index`, `list`

- *final return value*: `false` after the first item that failed to meet your criteria, else `true`
- **note**: stops iterating once it receives a falsy value from your callback.
- *example use case*:

```
const allPositiveNumbers = [1, 2, 3].every(function (item) {
  return item > 0;
});
console.log(allPositiveNumbers); // true
```

`find`:

- *callback is a predicate* - it should return a truthy or falsy value
- *callback answers*: is this item what you're looking for?
- *callback gets these arguments*: `item`, `index`, `list`
- *final return value*: the item you're looking for, or undefined
- **note**: stops iterating once it receives a truthy value from your callback.
- *example use case*:

```
const objects = [{ id: 'a' }, { id: 'b' }, { id: 'c' }];
const found = objects.find(function (item) {
  return item.id === 'b';
});
console.log(found === objects[1]); // true
```

`findIndex`:

- *callback is a predicate* - it should return a truthy or falsy value
- *callback answers*: is this item what you're looking for?
- *callback gets these arguments*: `item`, `index`, `list`
- *final return value*: the index of the item you're looking for, or `-1`
- **note**: stops iterating once it receives a truthy value from your callback.
- *example use case*:

```
const objects = [{ id: 'a' }, { id: 'b' }, { id: 'c' }];
const foundIndex = objects.findIndex(function (item) {
  return item.id === 'b';
});
console.log(foundIndex === 1); // true
```