Notes - JavaScript - Collections - Array

- Dr Nick Hayward

A brief introduction to collections, and arrays, with plain JavaScript.

Contents

Intro Collections in JS includes arrays, and associated built-in array methods, plus ES6 updates for *sets* and *maps*.

Arrays in JS are simply objects. So, arrays can access methods, like other objects.

Create arrays Two fundamental ways to create new arrays:

- using the built-in Array constructor
- using array literals `[]`

```
const countries = ["France", "Japan", "Canada"];
const cities = new Array("Tokyo", "Paris");
```

However, array literals tend to be the more common option for JS development.

n.b. writing to indexes outside the array bounds extends the array.

e.g. `countries.length === 5`

Likewise, if we try to write to a position outside of array bounds, as in

```
countries[4] = "Portugal";
```

the array will expand to accommodate the new inserted value.

We may end up creating a hole in the array, and the item at index 3 will be `undefined` . The length property will also be updated.

```
countries = ["France", "Japan", "Canada", "Portugal"];
```

Length of the array will now be 5, and the index positions will be as follows

- 0 = France
- 1 = Japan
- 2 = Canada
- 3 = empty
- 4 = Portugal

So, we now have a *hole* or *empty slot* in the array.

Adding and removing items at either end of an array A few simple methods we can use to add items to and remove items from an array:

- *push* - adds an item to the end of the array
- *unshift* - adds an item to the beginning of the array (existing items are moved forward one index posn)
- *pop* - removes an item from the end of the array
- *shift* - removes an item from the beginning of the array (existing items are moved back one index posn)

n.b. push and pop are faster than shift and unshift due to modifications of the index…

Adding and removing items at any array location If we simply delete an array item, we leave a hole at that index position with `undefined` …

Also, array length will still include this hole…

So, instead we need to use the 'splice' method for insertion and deletion.

For example,

```
var removedItems = countries.splice(1, 1);
```

This removes a single item at index posn 1.

The splice method will also return its own array of deleted items.

```
["Japan"]
```

Using the splice method, we can also insert items into arbitrary positions in an array.

For example, consider the following code:

```
var countries = ["France", "Japan", "Canada"];
removedItems = countries.splice(1, 2, "Spain", "Italy", "UK");
```

Starting from index 1, it first removes two items and then adds three items: "Spain", "Italy", and "UK".

The updated countries array is now as follows

```
["France", "Spain", "Italy", "UK"]
```

and the deleted arrays items will be available in the variable `removedItems`

```
["Japan", "Canada"]
```

Common operations on arrays Some common operations on JS arrays include,

- *iterate* - traverse arrays
- *map* - map existing array items to create a new array based on these items
- *test* - check array items match certain conditions
- *find* - find specific array items
- *aggregate* - compute a single value based on array items
  - e.g. compute total for array from array items…

iterate with `forEach` All JS arrays have a built-in method for `forEach` loops.

```
const archives = ['waldzell', 'mariafels'];

archives.forEach(archive => {
    console.log(`archive name = ${archive}`);
});
```

map arrays With array mapping, we're creating a new array based on the items in an existing array. This has become common usage in JavaScript development.

The idea is simple - we *map* each item from one array to a new item in a new array. So, we might extract just *names* from an array of archives.

```
// array
const archives = [
  {name: 'waldzell', type: 'game'},
  {name: 'mariafels', type: 'benedictine'}
];

// map array items to new array
const archiveNames = archives.map(archive => archive.name);

// iterate through new array
archiveNames.forEach(archive => {
  console.log(`archive name = ${archive}`);
});
```

test array items We may need to check one or more array items to see if they match certain conditions.

To help with this requirement, JS provides some useful built-in functions, `every` and `some` .

- `every` method - pass a callback, which is called for each specified property in the array
  - e.g. check if all properties have a specified value &c.
  - returns a boolean for the check - `true` for *all* properties matching specified value, otherwise `false`
- `some` method - pass a callback, which is called for each specified property in the array
  - e.g. check at least one property matches a specified value
  - returns a boolean - `true` for at least one match, `false` for zero matches

```
// array
const archives = [
  {name: 'waldzell', type: 'game', location: 'castalia'},
  {name: 'mariafels', type: 'benedictine'}
];

// check archives - `every` returns true for all match, `false` for single error/omission
const everyName = archives.every(archive => 'name' in archive);
// check boolean return for `every` method in everyName
everyName === true ? console.log(`each archive has a name`) :
                            console.log(`at least one archive is unnamed...`);

// check archives - `some` return true for a single match, `false` for no matches
const singleLocation = archives.some(archive => 'location' in archive);
// check boolean return value
singleLocation === true ? console.log(`at least one archive has a location`) :
                                      console.log(`no archive has a location...`);
```

searching arrays We can also search and find items in JS arrays. JS provides another built-in function, `find` .

```
// array
const archives = [
  {name: 'waldzell', type: 'game', location: 'castalia'},
  {name: 'mariafels', type: 'benedictine', location: 'czech'}
];

// find object in array
const locations = archives.find(archive => {
  // return object - not found simply returns undefined
  return archive.location === 'castalia';
});

// check search - check undefined or log archive name to console
locations !== undefined ? console.log(`archive in castalia = ${locations.name}`) :
                          console.log(`location and archive not found...`);
```

If the requested item can be found, the matching object will be returned. Otherwise, the `find` method will simply return `undefined` .

`find` will return the first matching item, regardless of the number of matches.

However, to search an array for all matches we can use the `filter` method instead.

```
// filter array and return multiple matches
const filterTypes = archives.filter(archive => 'type' in archive);
```

This will return all matching items, and we can simply check length of the return object, and iterate through the results.

```
// check filter returns
if (filterTypes.length >= 1 ) {
  for(let archive of filterTypes) {
    console.log(`archive name = ${archive.name} and type = ${archive.type}`);
  }
} else {
  console.log(`archive types are not available...`);
}
```

It's also possible to filter an array by index, using the following methods,

- `indexOf` = find the index of a given item, e.g.

```
const waldzellIndex = archives.indexOf('waldzell');
```

- `lastIndexOf` = find last index of multiple matched items, e.g.

```
const waldzellIndex = archives.lastIndexOf('waldzell');
```

- `findIndex` = effectively works the same as `find` but returns an index value, e.g.

```
const waldzellIndex = archives.findIndex(archive => archive === 'waldzell');
```

Resources

- MDN - JS - Array
- MDN - JS - Loops and Iteration