Array methods

`Array.from`

- Creates new, shallow-copied array instance from an array object
- Does not mutate the existing array

Goal: To understand the method does not mutate the original array, and the input is a string, but the output is a new array instance

- Example 1:

```
var arr1 = 'foo';
console.log(Array.from(arr1)); // ['f', 'o', 'o']
console.log(arr1); // 'foo'
var arr2 = [1, 2, 3, 'foo']
console.log(Array.from(arr2)); // [ 1, 2, 3, 'foo' ]
var arr3 = '1234foo';
console.log(Array.from(arr3)); // ['1', '2', '3', '4', 'f', 'o', 'o'];
```

`Array.isArray`

- Returns a boolean value whether the input is an array or not
- Goal: return value of this method is a boolean, and even though an array is an object, an object is not an array

Example 2:

```
Array.isArray([1, 2, 3]); // true
Array.isArray({foo:123}); // false
```

`Array.of()`

- Creates a new Array instance from a variable number of arguments, regardless of number or type of arguments
- The `Array` constructor method handles integer arguments (creates array of that length) a bit differently than the `Array.of()` method (creates array with a single element).

Goal: Understand that this method can turn any input into an array. But it's also very important to understand the difference between an `Array.of()` method and an `Array()` method.

When there is only one argument, both methods results in the creation of a new array, e.g. `Array(1, 2, 3)`, and `Array.of(1, 2, 3)` both return a new array `[1, 2, 3]`. The `Array()` method is very handy in creating empty slots for an array, which can be used to stuff an array of a certain length.

Example 3:

```
Array.of(7); // [7]
Array.of(1, 2, 3); //[1, 2, 3]

Array(7); // array of 7 empty slots
Array(1, 2, 3); // [1, 2, 3]
```

`Array.prototype.concat()` Goal: this method produces a new string, and neither originals are mutated in any way

Example 4:

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
console.log(array1.concat(array2));
```

`Array.prototype.copyWithin()`

- Copies part of an array to replace element at another index of the same array, mutating the original array in the process.
- Note that a piece of the array is copied and not cut, and it replaces (and not added) to the existing index of the arraycons

Goal: Understand 1. What each argument specifies, where the element is inserted, the two optional arguments that specify where the copying starts, and the second optional argument that specifies where the copying ends. 2. Understand the copying does not cut the elements in the current array, but when the elements get inserted, they do replace the current elements of the array. 3. All changes take place with the current array and it is mutated.

- Example 5:

```
const array1 = ['a', 'b', 'c', 'd', 'e']; // copy array elements starting from 3 to 4, insert that into index 0
console.log(array1.copyWithin(0, 3, 4)); //[ 'd ', 'b', 'c', 'd', 'e' ]
console.log(array1); // [ 'd', 'b', 'c', 'd', 'e' ]
console.log(array1.copyWithin(1, 3)); // // [ 'd', 'd', 'e', 'd', 'e' ]
console.log(array1); // [ 'd', 'd', 'e', 'd', 'e' ]
```

`Array.prototype.entries()`

- This method returns a new Iterator object that containers the key-value pair for each index in the array. Note that the object will always take the index of the array and the corresponding object. It's a great way to turn an array into an iterator that has its index attached.
- How to use it? See the two examples below, whereas the `for()` loop typically can be used to grab each element of an array, in the case of an iterator, two parameters are made available, both the index of the array element and the array element itself. The syntax then becomes `for([index, element] of array.entries())`.

Goals: to understand what kind fo Object `#entries()` turns an array into, and the arguments then made available to it.

- Example 6:

```
var a = [ 'a', 'b', 'c' ];

for (e of a.entries()) {
  console.log(e);
}
// Output:
[ 0, 'a' ]
[ 1, 'b' ]
[ 2, 'c' ]
```

- Example 6b:

```
var a = [ 'a', 'b', 'c' ];

for (const [index, element] of a.entries()) {
  console.log(index, element);
}
// Output:
// 0 'a'
// 1 'b'
// 2 'c'
```

`Array.every()`

- Return a boolean value that reflects whether every single element in an array passes a test implemented by the function
- Goal: Understand that `#every` checks for every single element and returns `true` if only all is true. And that the argument that the method takes could very well be a function expression defined elsewhere.
- Example 7:

```
const isBelowThreshold = (currentValue) => currentValue < 40;
const array3 = [1, 30, 39, 29, 10, 13];
console.log(array3.every(isBelowThreshold)); // true
```

`Array.prototype.fill()`

- Changes all elements in an array to a static value, from start 0 to end index, and returns the modified array. This is somewhat similar to the `Array.prototype.copyWithin()` method, except one can specify what needs to be filled, instead of using what already exists in the array.
- Note that the first argument is required, and the other two are not
- The first argument specifies what is going to fill the array
- The second and third arguments are optional, and specifies the from...to positions of the fill
- If the last argument isn't given, then it's assumed the end index is the length of the array

Goal: To understand the input that is optional versus mandatory, and the output

- Example 8:

```
const array1 = [1, 2, 3, 4];
console.log(array1.fill(0, 2, 4)); // [1, 2, 0, 0]
```

`Array.prototype.filter()`

- Filters an array based on input or a function expression. This is similar to a number of Array methods like `#every()` and `#any()`, except the output is an array instead of a boolean value.
- Example 9:

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destrubtion', 'present'];
const filterCondition = word => word.length > 6;
console.log(words.filter(filterCondition)); ['exuberant', 'destrubtion', 'present']
```

`Array.prototype.find()`

- Very similar to how `#every` and `#any` works.
- Example 10:

```
const array1 = [5, 12, 8, 130, 44];
const findCondition = el => el > 10;
console.log(array1.find(findCondition)); // 12
```

`Array.prototype.findIndex()`

- This method finds the index of an element based on a defined condition
- Works almost exactly the same as `Array.prototype.findIndex()`
- Example 11:

```
const array1 = [5, 12, 8, 130, 44];
const findIndexCondition = el => el > 13;
console.log(array1.findIndex(findCondition)); // 3
```

`Array.prototype.flat()`

- This method creates a new array with all sub-array elements concatenated to it recursively up to the specified path. In order words, it flattens the array to the level specified in the argument. `Infinity` would collapse all the subarrays within the input array.
- Example 12:

```
var arr1 = [1, 2, [3, 4]];
arr1.flat(); // [1, 2, 3, 4]
var arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat(); // [1, 2, 3, 4, [5, 6]];
arr2.flat(2); // [1, 2, 3, 4, 5, 6];
arr4.flat(Infinity); // [1, 2, ... 10];
```

`Array.prototype.flatMap()`

- Example 13:

```
let arr1 = [1, 2, 3, 4];
var mapCondition1 = x => [x * 2];
var mapCondition2 = x => [[x * 2]];
arr1.map(mapCondition1); // [ [ 2 ], [ 4 ], [ 6 ], [ 8 ] ]
arr1.flatMap(mapCondition1); // [ 2, 4, 6, 8 ]
arr1.flatMap(mapCondition2); // [ [ 2 ], [ 4 ], [ 6 ], [ 8 ] ]
```

`Array.prototype.includes()`

- Example 14:

```
const array1 = [1, 2, 3];
console.log(array1.includes(2)); // true
```

`Array.prototype.indexOf()`

```
const beasts = ['ant', 'bison', 'camel', 'duck', 'bison'];
console.log(beasts.indexOf('bison'));
```

Array.prototype.keys()

- Example 15:

```
const array1 = ['a', 'b', 'c'];
const iterator = array1.keys();
for (const key of iterator) {
  console.log(key);
} // 0, 1, 2

var arr = ['a', 'c'];
var sparseKeys = Object.keys(arr);
var denseKeys = [...arr.keys()];
console.log(sparseKeys); [ '0', '1' ], picks up the keys as strings
// or, console.log(sparseKeys.map(key => +key));
console.log(denseKeys); [ 0, 1 ], picks up the keys as integers
```

Array.prototype.lastIndexOf()

- This returns the last index of an item searched for in an array
- Example 16

```
const animals = ['Dodo', 'Tiger', 'Penguin', 'Dodo'];
console.log(animals.lastIndexOf('Dodo')); // 3
console.log(animals.indexOf('Dodo')); // 0
```

Array.prototype.map()

- Example 17

```
const array1 = [1, 4, 9, 16];
const map1 = array1.map(x => x * 2);
console.log(map1); // [2, 8, 18, 32]

var kvArray = [ {key: 1, value: 10},
                {key: 2, value: 20},
                {key: 3, value: 30}];

var reformattedArray = kvArray.map (
  obj => {
    var rObj = {};
    rObj[obj.key] = obj.value;
    return rObj;
  });

reformattedArray;
// [ { '1': 10 }, { '2': 20 }, { '3': 30 } ]
```

Array.prototype.pop()

- Returns the last element in an array, original array is mutated
- Example 18

```
const letters = ['a', 'b', 'c', 'd'];
console.log(letters.pop());
console.log(letters);
```

Array.prototype.push()

- Pushes in an element at end of array and mutates the array, returns the length of array
- Example 19

```
const letters = ['a', 'b', 'c'];
console.log(letters.push('d'));
```

```
console.log(letters);
```

`Array.reduce()`

- Takes a reducer function that executes on each element of an array, resulting in a single output
- Example 20

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

console.log(array1.reduce(reducer));
console.log(array1.reduce(reducer, 5));
```

`Array.reduceRight()`

- Reduces an array from right to left based on what's specified in the reducer function
- Example 21

```
var reduceCondition = (accumulator, currentValue) => accumulator.concat(currentValue);
var reduceCondition2 = (accumulator, currentValue) => accumulator + currentValue;

const array = [[0, 1], [2, 3], [4, 5]];
console.log(array.reduceRight(reduceCondition)); //  [4, 5, 2, 3, 0, 1 ];
console.log(array.reduceRight(reduceCondition2)); //  '4,52,30,1'

var array1 = ['a', 'b', 'c'];
console.log(array1.reduceRight(reduceCondition)); // cba
console.log(array1.reduceRight(reduceCondition2)); // cba

var array2 = [['a'], ['b'], ['c']];
console.log(array2.reduceRight(reduceCondition)); // [ 'c', 'b', 'a' ]
console.log(array2.reduceRight(reduceCondition2)); // 'cba'
```

`Array.prototype.reverse()`

- Reverses the array in place
- Example 22

```
var arr1 = ['one', 'two', 'three'];
console.log(arr1); // ['one', 'two', 'three']
console.log(arr1.reverse()); // ['three', 'two', 'one']
console.log(arr1); // ['three', 'two', 'one']
```

`Array.prototype.shift()`

- Example 23

```
var arr 1 = [1, 2, 3];
arr1.shift(); // 1
arr1; // [2, 3]
```

`Array.prototype.slice()`

- Original array not modified
- Returns shallow copy of array into new array object selected from begin to end
- Example 24

```
var arr = ['a', 'b', 'c', 'd', 'e'];
console.log(arr.slice(2)); // ['c', 'd', 'e'];
console.log(arr.slice(2, 4)); // ['c', 'd'];
console.log(arr.slice(1, 5)); // ['b', 'c', 'd', 'e'];
```

`Array.prototype.some()`

- Similar to `every()`, except this return true when one satisfies the condition

```
const array = [1, 2, 3, 4, 5];
const even = element => element % 2 === 0;
console.log(array.some(even);
```

```
Array.prototype.sort()
```

- This sorts the array in place and returns the sorted array
- Example 25: sorting integers defaults to sorting by string

```
var arr = [1, 30, 40, 21, 10000];
arr.sort(); // [ 1, 10000, 21, 30, 40 ];
```

- Example 26: sorting numbers by specifying how something should be sorted through a function

```
function sortCondition (a, b) {
  if (a < b) {
    return -1;
  } else if (a > b) {
    return 1;
  } else {
    return 0;
  }
}

var arr = [1, 30, 40, 21, 10000];
arr.sort(sortCondition); // [ 1, 21, 30, 40, 10000 ]
```

- Example 27: another way to specify integer sorting

```
function sortCondition1(a, b) {
  return a - b;
}

var arr = [1, 30, 40, 21, 10000];
arr.sort(sortCondition1); // [ 1, 21, 30, 40, 10000 ]
```

- Example 28: Sorting objects

```
var items = [
            {name: 'Shane', value: 37},
            {name: 'Ed', value: 21},
            ]

function sortCondition (a, b) {
  return a.value - b.value;
}
items.sort(sortCondition);
```