# Array methods

Seeing the latest HTTP 203 from Zurma and Jakke I started thinking what other things that you can do with arrays.

The most frequent use for arrays in my code is to take an array of elements and do something to each member of the array.

For example, the following snippet converts the `document.images` collectioon into an array and then uses `forEach` to loop over the array, and add the loading attribute to enable native lazy loading in Chrome and provide an alternative lazy loading library for browsers that don't support the feature natively.

```
if ('loading' in HTMLImageElement.prototype) {
  const myImages = [...document.images];

  myImages.forEach((myImage) => {
    myImage.setAttribute('loading', 'l'loading'})
} else {
  console.log('native lazy loading not supported');
  'native lazy loading not supported'// Fetch and apply a polyfill
  // for lazy-loading instead.
}
```

I know, if I'm creating the page from scratch it's better to add the attribute directly to the HTML but I'm lazy and adding the attribute by hand in image/video heavy pages can take longer than inlining the script on the head of the document.

It got me thinking, what else can we do with arrays and how much would it simplify my code.

In this post we'll explore some things you can do with and to arrays.

# Array.map

The `map` method creates a new array wher the content is the result of applying a function to the elements in the array.

```
const array1 = [1, 4, 9, 16];

const map1 = array1.map(x => x * x);

console.log(map1);
```

# Array.from

The `from` method creates a new, shallow-copied Array instance from an array-like or iterable object.

```
console.log(Array.from('foo'));
```

# Array.every

The `every` method tests whether all elements in the array pass the test in the test in the associated function. It returns a Boolean value.

```
const isAboveThreshold = (currentValue) => currentValue > 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isAboveThreshold));
```

# Array.filter

The `filter` method creates a new array with all elements that pass the test in the function we pass as the parameter.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'pre

const result = words.filter(word => word.length > 6);
```

```
console.log(result);
```

# Array.flat

The `flat` method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

In the example below, play with the value for myArray.flat and see how it changes the items in the array.

```
const myArray = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]];
myArray.flat(Infinity);
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Array.includes

The includes() method determines whether an array includes a certain value among its entries, returning true or false as appropriate.

Not in the example below that eventhough the string `at` appears twice in the array, includes returns false in the query. It appears that it's searching for full strings, not portions of one.

```
const pets = ['cat', 'dog', 'bat'];
console.log(pets.includes('cat'));

console.log(pets.includes('at'));
```

Arrays have many other methods for you to research and play with. These are the ones I find the most useful.