Arrays II & callbacks

A common array method is the `.filter()` method. When you call this method on an array, you will get back another array that contains some of the items from the original array, based on the condition you specify. Let's take an example:

```
let numbers = [9, 5, 14, 3, 11];

let numbersAboveTen = numbers.filter(function(number) {
    return number > 10;
});
console.log(numbersAboveTen); // [14, 11]
```

Don't forget the `return` keyword inside the callback function.

Notice how we got back a new array that contains the items which have satisfied the condition. The condition is that the `number` must be above 10.

Array.filter(callback)

The `.filter()` method expects a callback as the first argument. In our example, the callback is:

JavaScript will take your callback and call it for **every single item** in the array. Our `numbers` array has 5 items, so it will call it 5 times. Every time that it calls this function, it will give a value to the `number` parameter that you specified inside this callback.

- The first time it runs, it will give the `number` a value of 9 (the first item of the array).
- The second time it runs, it will give the `number` a value of 5 (the second item of the array).
- and so on and so forth until the last item of the array.

This is how callbacks work. Now every array method has a different behavior which we'll be explaining. This behavior often depends on the result of the callback. In this example, if the callback function returns `true`, then the item will be included in the final array returned by `.filter()`. However, if the callback function returns `false`, then the item will **not** be included in the final array.

This will return **every** item in the array. So you will end up getting a copy of the original array. That's because the callback is always returning true. This code is not very useful, but it's to show you the importance of what the callback function returns and how that affects the result of the `.filter()` method.

This is why we had a condition `number >= 10`. This condition will return either `true` or `false` depending on the value of the `number`.

JavaScript will take your callback and pass the item of the array as the **first parameter** to your callback function. This means that the code below works (but is not recommended):

```
let numbers = [9, 5, 14, 3, 11];

// works but is NOT recommended
let numbersAboveTen = numbers.filter(function(x) {
    return x > 10;
});
console.log(numbersAboveTen); // [14, 11]
```

JavaScript doesn't care about what you call your variables. It will call your callback function and give a value to the first parameter which we called `x` here.

However, from a developer perspective what is `x`? It's not clear at all so always make sure to follow the **plural -> singular** naming convention that we covered in the previous chapter. It will make your life easier.

The code below:

```
let years = [2000, 2008, 2020, 2023];

years.filter(function(year) {
    return year >= 2010;
});
```

Recap

- The .filter() method returns a new array that contains some of the items from the original array, based on the condition you specify.
- JavaScript will take your callback function and call it for every single item in the array.
- For the .filter() method, the result of the callback function matters. When it's true, the item will be included in the resulting array. Otherwise, it won't.
- JavaScript cannot make a smart guess that the numbers array becomes the number parameter in your callback function. What it does is that it calls your callback function while giving a value for the first parameter that you specified.
- Use the plural -> singular naming convention when using the .filter() method.

```
.find()

let names = ["Sam", "Alex", "Charlie"];

let result = names.find(function(name) {
  return name === "Alex";
});
console.log(result); // "Alex"
```

When you call the `.find(callback)` method on an array, you will get back **the first item** that matches the condition that you specify. If no items were found, you will get back `undefined`.

The condition that we specified here is that the `name` should be equal to `"Alex"`.

So the `.find(callback)` method will call the callback that you provided for every item in the array until one of the callbacks returns `true`. When this happens, it will stop calling the remaining callbacks and return to you the item for which the callback returned `true`.

In our example above, here's the callback:

```
function(name) {
  return name === "Alex";
}
```

which gets called for `name = "Sam"` (first item of the array). However, the callback will return `false` because `name === "Alex"` returns `false`. So the callback will be called again with the next value of name. This time, `name = "Alex"`. The callback will return `true` because `name === "Alex"` (the condition inside the callback) returns `true`. So the `.find()` method stops and returns to you that item which is `"Alex"`.

Let's take another example but this time with an array of numbers:

```
let numbers = [9, 5, 14, 3, 11];

let firstNumberAboveTen = numbers.find(function(number) {
    return number > 10;
});
console.log(firstNumberAboveTen); // 14
```

Notice how even though there are 2 numbers that satisfy the condition, the `.find()` method stops at the **first** one that satisfies the condition.

This will bring us to the next section, which is `.filter()` vs `.find()`. What are the differences?

.filter() vs .find()

So, what is the difference between `.filter()` and `.find()`?

The difference has to do with the **return type** of these 2 methods:

The `.filter()` method **always** returns an array.

Let's take a look at a few examples:

```
let numbers = [9, 5, 14, 3, 11];

// .filter() ALWAYS returns an array
numbers.filter(function(number) {
    return number >= 12;
}); // [14]

// .find() returns the first match or undefined
numbers.find(function(number) {
    return number >= 12;
}); // 14
```

Notice how the `.filter()` is returning an array, even if there's only 1 item that matches your condition. In contrast, the `.find()` method will return the first item that matches the condition.

.filter() always returns an array. Even if it matched one item or no items.

Now let's take a look at an example where no items satisfy the condition:

```
let numbers = [9, 5, 14, 3, 11];

// filter() ALWAYS returns an array (even if it's empty)
numbers.filter(function(number) {
    return number >= 15;
}); // []

// .find() returns the first match or undefined (when none of the items satisfy the condition)
numbers.find(function(number) {
    return number >= 15;
}); // undefined
```

Notice how the `.filter()` returned an empty array and the `.find()` returned `undefined`.

`.find(callback)` can return `undefined`. You may have to wrap its result in an `if condition` to avoid unexpected errors if you end up calling a method on its result.

Recap

- The .find() method returns the first item which matches the condition that you specify. If no items were found, you will get back undefined.
- The .filter() method always returns an array. Even if it's empty.

Array map

The `.map(callback)` method allows you to **transform** an array into another one. Here are some common examples:

`[4, 2, 5, 8]` transformed to `[8, 4, 10, 16]`. We doubled every item in the original array.

`["sam", "Alex"]` transformed to `["SAM", "ALEX"]`. We upper cased every item in the original array.

Notice that you always get back an array containing the **same number of items** compared to the original array, but every item has most likely undergone some transformation

In the first example, the transformation is that we multiply every number by 2. In the second example, the transformation is that we call `.toUpperCase()` on every item.

Let's take a look at how we can implement these transformations:

```
const numbers = [4, 2, 5, 8];

const doubled = numbers.map(function(number) {
    return number * 2;
});
console.log(doubled); // [8, 4, 10, 16]
```

and

```
const names = ["sam", "Alex"];
const upperNames = names.map(function(name) {
    return name.toUpperCase();
});
```

If you forget the `return` inside the callback function, you will end up with the following array: `[undefined, undefined]`. That's because, for every item in the original array (`["sam", "Alex"]`), you're returning `undefined` so the end result will be `[undefined, undefined]`.

Array includes(item)

The array `.includes(item)` method is one of the simplest array methods as it takes an `item` rather than a callback and returns `true` when that `item` exists in the array and `false` otherwise. Here's an example:

```
const groceries = ["Apple", "Peach", "Tomato"];

groceries.includes("Tomato"); // true
groceries.includes("Bread"); // false
```

Array join(glue)

When you have an array and you render this array to a web page (as we'll see later on in the DOM section of the course), the array will be automatically converted to a string. JavaScript will automatically invoke the `.toString()` method of the array which returns a string of the array elements separated by commas. Here's how it works:

```
const groceries = ["Apple", "Peach", "Tomato"];
groceries.toString(); // "Apple,Peach,Tomato"
```

But there's a downside, which is that you cannot customize the glue that gets inserted in between the array items, which is the comma `,` character.

If you'd like to customize the glue, then you can use the `.join(glue)` method:

```
const groceries = ["Apple", "Peach", "Tomato"];
groceries.join("; "); // "Apple; Peach; Tomato"
groceries.join(" . "); // "Apple . Peach . Tomato"
```

Recap

- The array .map(callback) method allows you to transform an array into another one.
- The array .includes(item) method takes an item and returns true when that item exists in the array and false otherwise.
- The array .join(glue) method returns a string of the array elements separated by the glue.

Chapter Recap

- The .filter() method returns a new array that contains some of the items from the original array, based on the condition you specify.
- JavaScript will take your callback function and call it for every single item in the array.
- For the .filter() method, the result of the callback function matters. When it's true, the item will be included in the resulting array. Otherwise, it won't.
- JavaScript cannot make a smart guess that the numbers array becomes the number parameter in your callback function. What it does is that it calls your callback function while giving a value for the first parameter that you specified.
- Use the plural -> singular naming convention when using the .filter() method.
- The .find() method returns the first item which matches the condition that you specify. If no items were found, you will get back undefined.
- The .filter() method always returns an array. Even if it's empty.
- The array .map(callback) method allows you to transform an array into another one.
- The array .includes(item) method takes an item and returns true when that item exists in the array and false otherwise.
- The array .join(glue) method returns a string of the array elements separated by the glue.

Show previous notes