{ Array Methods. }

Objectives

By the end of this chapter, you should be able to:

- Use the array's length to see how many elements the array has
- Use array modification methods like `join`, `slice`, `concat` to modify arrays or create new arrays.
- Search for elements in an array using `indexOf` and `lastIndexOf`
- Describe the difference between a reference type and a value type

So far, we've seen how to access, update, add and remove items from an array. We've also encountered some common array methods, including `push`, `pop`, `shift`, `unshift`, and `splice`. But these aren't the only methods you're likely to encounter when working with arrays. Let's take a look at a few more.

Common array functions and properties

length

`length` returns how many elements are in the array. This is a property, NOT a function (you can tell because we type `length`, not `length()`. As we've seen, it can (but is almost never) be used to remove elements/clear an array.

```
let arr = [1,2,3,4];
arr.length; // 4
arr[arr.length]; // undefined
arr[arr.length-1]; // 4 - this is a nice way to access the last element of an array w
hen you don't know how many elements are inside it.
```

slice

`slice` makes a copy of an array. We can use it to copy the entire array, or create a copy of a *subarray*. If we just invoke `slice()` with no arguments, we'll create a copy:

```
let arr = [1,2,3,4];
let copy = arr.slice();
copy; // [1,2,3,4];
```

Alternatively, you can pass in two arguments to `slice`. Like `splice`, the first argument indicates the starting index of the subarray you want. The second argument indicates the ending index. The subarray you get will consist of all the values starting from the starting index and going up to (but **not including**) the ending index:

```
let arr = [7, 6, 5, 4, 3, 2];
arr.slice(1, 2); // [6]
arr.slice(2, 5); // [5, 4, 3]
arr.slice(2, 1); // []
```

concat

`concat` joins two arrays together.

```
let arr1 = [1,2,3];
let arr2 = [4,5,6];
let combined = arr1.concat(arr2);
combined; // [1,2,3,4,5,6]
```

In fact, you can pass multiple arrays into `concat` and it will still return a single array to you:

```
let arr1 = ["a","b","c"];
let arr2 = ["d","e","f"];
let arr3 = ["g","h","i"];
let combined = arr1.concat(arr2,arr3);
combined; // ["a","b","c","d","e","f","g","h","i"];
```

What's more, you don't even need to pass an array into `concat`! Any comma-separated list of values can be concatenated with the original array:

```
let openingWords = ["It","was","a"];
let moreOpeningWords = openingWords.concat("dark","and","stormy","night");
moreOpeningWords; // ["It", "was", "a", "dark", "and", "stormy", "night"]
```

join

`join` joins elements of an array into a string separated by whatever you pass in as an argument to `join`. This argument is frequently referred to as a *delimiter*. Here are a couple of examples:

```
let arr = ["Hello", "World"];
arr.join(" "); // "Hello World"

let arr2 = ["I", "have", "a", "big", "announcement"];
arr2.join("! ") + "!"; // "I! have! a! big! announcement!"
```

indexOf

`indexOf` finds the first index of the element passed in (starting from the left). If the element is *not* found, it returns -1. Here are some examples:

```
let arr = [1,2,3,4,5,4,4];
arr.indexOf(2); // 1
arr.indexOf(3); // 2
arr.indexOf(1); // 0 - remember, arrays are zero indexed
arr.indexOf(4); // 3 - indexOf stops once it finds the first 4.
arr.indexOf(10); // -1
```

You'll see this function very commonly used to check if an element is in an array or not. Here's an example:

```
let moviesIKnow = [
    "Wayne's World",
    "The Matrix",
    "Anchorman",
    "Bridesmaids"
];

let yourFavoriteMovie = prompt("What's your favorite movie?");
if (moviesIKnow.indexOf(yourFavoriteMovie) > -1) {
    alert("Oh, cool, I've heard of " + yourFavoriteMovie + "!");
} else {
    alert("I haven't heard of " + yourFavoriteMovie + ". I'll check it out.");
}
```

lastIndexOf

`lastIndexOf` works just like `indexOf`, but starts searching from the end of the array rather than the beginning.

```
let arr = [1,2,3,4,5,4,4];
arr.indexOf(4); // 3
arr.lastIndexOf(4); // 6 - this one is different now as it starts from the end!
arr.lastIndexOf(10); // -1 - still returns -1 if the value is not found in the array
```

Reference vs Value

An essential distinction between primitives and objects (including arrays, which are a type of object in JavaScript) is how their values are passed when assigned to new variables. Take a look at the following example:

```
let instructor = "Elie";
let anotherInstructor = instructor;
anotherInstructor // "Elie";

// Let's assign a new value to anotherInstructor:
anotherInstructor = "Matt";

instructor; // "Elie"
anotherInstructor; // "Matt"
```

In this example, even though we changed the `anotherInstructor` variable, it did not affect the `instructor` variable. This is because each one of these primitive types has a specific address in memory (it is a bit more complex than that, but we'll keep things simple to start). Another way to think of this is that when we assigned `anotherInstructor` to equal `instructor`, JavaScript created a copy of the string "Elie" and assigned that value to `anotherInstructor`. So even though those two variables were storing identical-looking strings, they can be modified independently of one another.

This may seem confusing until we compare this with what happens when dealing with reference types. Let's take a look at this array:

```
let instructors = ["Elie", "Matt"];
let instructorCopy = instructors;
```

```
instructors === instructorCopy // true
instructorCopy.push("Tim");

instructorCopy; // ["Elie", "Matt", "Tim"]
instructors; // ["Elie", "Matt", "Tim"]
```

We see here that the `original` instructor array was changed when we pushed Tim to `instructorCopy`! This is because the `instructorCopy` did not create a new array, it just created a reference (or pointer) to the `instructors` array. In other words, unlike with our previous example, setting `instructorCopy` equal to `instructors` doesn't create a *copy* of the `instructors` array in JavaScript. Instead, both variable names refer to the exact same array! You also notice here that we are comparing two arrays using `===` and it evaluates to `true`. When comparing two arrays (and objects) using `===`, that expression will always evaluate to false unless they are the same reference.

```
let instructors = ["Elie", "Matt"];
let instructorCopy = instructors;

instructors === instructorCopy // true

let instructorsAgain = ["Elie", "Matt"];
instructors === instructorsAgain // false (not the same reference)

let instructorsCopyWithSlice = instructors.slice()
instructors === instructorsCopyWithSlice // false (not the same reference)
```

This can take some time to wrap your head around. If you're curious, you can read more about the phenomenon of passing by value vs. passing by reference here and here

Exercises

Complete both parts of the exercises before moving on. It is **essential** that you complete these exercises as they are the building blocks for the next section!

Part I

Write the code necessary to do the following:

- Create an empty array called `arr`.
- Add your first name to the `arr` variable

- Add your last name to the end of the `arr` variable
- Add your favorite color to the beginning of the `arr` variable

Your variable `arr` should look like this (using Elie for a first name, Schoppik for a last name and purple for a favorite color) `["purple", "Elie", "Schoppik"]`. Keep going!

- Remove the favorite color from the `arr` variable (remember this is the first value in the array - what method can you use to remove the first value in an array?)
- Create another array called `arr2`.
- Add your favorite number to `arr2`
- Add the string "JavaScript" to the **end** of the `arr2` variable

Your variable `arr2` should look like this (using 42 as a favorite number) `[42, "JavaScript"]`.

- See if the value `42` exists in the `arr2` array. Do this using the `indexOf` method. What does `indexOf` return to you if the value passed to it can not be found in the array?
- Create a new variable called `combinedArr` which is the result of your `arr` and `arr2` variables combined into one array.

Your `combinedArr` variable should look like this (using our previous values) `["Elie", "Schoppik", 42, "JavaScript"]`

Part II

Complete the following, starting from the following array: `let arr = ["JavaScript", "Python", "Ruby", "Java"]`

1. Return the following array: `["Python", "Ruby"]`.
2. Combine the array with the array `["Haskell", "Clojure"]`.
3. Return the string `"JavaScript, Python, Ruby, Java"`.
4. Try to explain, in your own words (or diagrams!) what the difference is between passing by value vs. passing by reference.

Before you continue!

Before you move on, check your understanding and make sure you can answer yes to the following questions:

- Do I know how to create an array?
- Do I know how to add an element to the beginning of an array?
- Do I know how to add an element to the end of an array?
- Do I know how to remove an element from the end of an array?
- Do I know how to remove an element from the beginning of an array?
- Do I know how to remove an element from a specific index in an array?
- Do I know how to add one or more elements at a specific index in an array?
- Do I know what the `includes` function does?
- Do I know what the `indexOf` function does?
- Do I know the difference between `includes` and `indexOf`?
- Do I know why `[] === []` always returns `false`?