Arrays

WE'LL COVER THE FOLLOWING

Arrays in ES6 have also gotten a little love. There are quite a few new methods available for use on them, and there is also a brand new way to iterate over arrays. In JavaScript there is something called Iterator Objects, these are things like arrays, strings and some things we haven't looked at yet called Map and Set. In ES6 the `for...of` loop is a new construct that allows us to iterate over these objects.

for...of

The syntax for the `for...of` loop is very similar to that of the `for...in` loop, but instead of looping through the keys of an object it will loop through the elements of the array.

```
let programmingLangauges = ['JavaScript','Ruby','GO'];
```

```
for (let language of programmingLangauges) {
    console.log('I really like ' + language);
}

//"I really like JavaScript"
//"I really like Ruby"
//"I really like GO"
```

There is not much else to look at here. The real benefit is that we don't have to create a `for` loop the old-fashioned way.

```
let programmingLangauges = ['JavaScript','Ruby','GO'];

for (let i = 0; i < programmingLangauges.length; i++) {
    console.log('I really like ' + programmingLangauges[i]);
}
```

This does the same thing as the above, but it is a little more verbose, and we have to configure the loop. If we simply just want to loop over an array, the `for...of` loop is great for that!

Typed Arrays

Something I wanted to briefly look over are Typed Arrays. In ES6 you now have the ability to create arrays that are made up of just elements of a specific data type. JavaScript is a dynamically-typed language, meaning that when we define a variable we don't have to define that it will be a string or number. The reason for creating Typed Arrays is to create data that will perform better than the average array. This is typically for data used in WebGL programs or something that has to deal with binary data.

There are a few ways to create a Typed Array.

```
let newTypedArray = new Int8Array(5);
newTypedArray[0] = 1;
newTypedArray[1] = 2;
newTypedArray[2] = 3;
```

You can do it by using the `new` keyword and using one of the available types

`Int8Array`, `Int16Array`, `Int32Array`, etc. You also have to the length that would

like your array to be in the constructor. To find out more check out the MDN article.

Another way that we can create a Typed Array is by passing the values directly into the array type that you want to create.

```
let newTypedArray = new Int8Array([1,2,3]);
```

This will do that same as the above. But, what is the point of using a typed array? When an interpreter know that the values it will encounter will never change from a specific type, it can do some optimizations for that! And this makes things fast.

Array.from()

The `Array.from` method allows us to take an array-like object or Iterator Object and return an actual array from this. Let's look at some examples!

The most useful example is when working with the `arguments` objects. This is a value that we have access to when we call a function (not an arrow function though).

```
function add() {
    console.log(arguments);
}
add(100, 200, 300); //{ '0': 100, '1': 200, '2': 300 }
```

The problem here is that it's not an actual array (we will solve this problem another way later in the book). We can, however, use `Array.from` to turn this into an array. The real issue here is that we can't use an array method like `.reduce()` to sum up these numbers.

```
function add() {
    args = Array.from(arguments);
    return args.reduce(function(num1,num2) { return num1 + num2 }, 0);
}
```

```
let result = add(100,200,300);
console.log(result); //600
```

We can also do this with stuff like strings. Strings are what we call an Iterator Object.

```
let arrayFromString = Array.from('This is neat!');
console.log(arrayFromString); //["T", "h", "i", "s", " ", "i", "s", " ", "n", "e", "a", "t",
```

`Array.from` will also accept another argument, a map function used to map over the provided array-like/iterable object.

```
function double() {
    return Array.from(arguments,function(n) { return n * 2 });
}
console.log(double(1,2,3)); //[2,4,6];
```

Array.of()

The `Array.of` method is a simple method that allows us to create an array of values. The syntax looks like this:

```
const numbersArray = Array.of(1,2,3,4,5,6);
```

It creates an array OF the values passed to it. There is not much else to it!

find() & findIndex()

If you have ever used a library like Underscore or Lodash, you have probably used a method like `_.find()` that will allow you to search through an array and find the first instance of a value. Let's look at an example and assume we have a set of data like this:

```
const students = [
    {
        name: 'Steve',
        course: 'History'
    },
    {
        name: 'Mary',
        course: 'Science'
    },
    {
        name: 'Lisa',
        course: 'Physics'
    },
    {
        name: 'Michelle',
        course: 'Physics'
    }
];
```

In this array of student objects, perhaps we want to find a student that is taking Physics. Using `Array.find` we can find the student like this.

```
const physicsStudent = students.find(function(student) {
    return student.course === 'Physics';
});

console.log(physicsStudent);
/*
{
    course: 'Physics',
    name: 'Lisa'
}
*/
```

The `.find` method will find the first instance of your search: you provide the method with a callback function that is passed each element from the array. The callback returns a check, in this case what we want to be true, and only once the condition `student.course === 'Physics'` is true will the `.find` method stop and return that element from the array.

It's important to remember that this will just find ONE instance of the search. If you want to get multiples, consider the `.filter` method.

The `.findIndex` method is pretty much identical, however instead of it returning the element that matches the check, it will return the index of that element in the array.

```
const physicsStudent = students.findIndex(function(student) {
    return student.course === 'Physics';
});

console.log(physicsStudent); //2
```

`.find` and `.findIndex` also accept an optional second argument: an object that should be used as the `this` in the callback if needed.

Values, Keys & Entries

These next three methods are similar to each other, so it makes sense to talk about them together. Each of these methods will return a new Iterator Object; we will explore these a little more when we get to the `Map` and `Set` chapters. But, an Iterator Object is an object that implements a `.next` method. This is a method we can use to iterate through values!

values()

The `.values` method will take an array and return an object that will allow us to step through each value individually. Let's keep using the `students` object from above for our examples here.

```
const studentValues = students.values();

console.log(studentValues.next().value);
/*
{
    name: 'Steve',
    course: 'History'
}
*/
console.log(studentValues.next().value);
/*
{
    name: 'Mary',
    course: 'Science'
}
*/
```

The `.next` method will let you iterate, one at a time, through the values in the array! We could also use this in a `for...of` loop if we wanted, however that would just be very similar to passing an array to the `for...of` loop.

keys()

Similar to `.values`, the `.keys` method will return an Iterator Object, but this time the values returned are the index number for the current iteration you are on.

```
const studentKeys = students.keys();

console.log(studentKeys.next().value); //0
console.log(studentKeys.next().value); //1
console.log(studentKeys.next().value); //2
```

Again we could pass this to a `for...of` if needed. One thing to note is that when you exhaust the next available iteration of an these Iterator Objects, it will return `undefined` as a value.

> sentence needs clarification

entries()

Last is the `.entries` method which is very similar to the first two. However, the values returned from this method are in an array. From this array the first element is the index of the element and the second is its value.

```
const studentEntries = students.entries();

console.log(studentEntries.next().value);
/*
[0, {name: 'Steve',course: 'History'}]
*/
console.log(studentEntries.next().value);
/*
[1, {name: 'Mary',course: 'Science'}]
*/
```

The `.values`, `.keys` and `.entries` methods all return Iterator Objects that allow us to step through the values, each with a different version of the data. We will see more Iterator Objects in chapters to come!

copyWithin()

The `.copyWithin` method provides a way for you to take a portion of your array and copy it to a new location in the same array. It takes three arguments, the target index to copy too, the start index, and the end index.

An example will help explain it best:

```
const nameArray = ['R','Y','A','N'];

console.log(nameArray.copyWithin(0,2,3)); //['A','Y','A','N'];
```

In the example above, we have an array of letters, using the `.copyWithin` method I can say this. I want to target the first index(`0`) then start at the third index(`2`) and copy up to but not including the fourth index (`3`). Then place that in the target location `0`. If I wanted to have the first letter be `N` in the array I could change the `.copyWithin` call to look like this.

```
const nameArray = ['R','Y','A','N'];

console.log(nameArray.copyWithin(0,3,4));
```

It is important to note that since `4` is not an index in the array, it will just go up until the end of the array's length. You may also use negative numbers here as well.

```
const nameArray = ['N','Y','A','N'];

console.log(nameArray.copyWithin(-1,1,2));//["N", "Y", "A", "Y"]
```

The `-1` here represents the the `'N'` or last element in the array, the negative number will just loop back around. In this case it will place the `Y` at the end.

Why use copyWithin?

The `.copyWithin` method is a method that is mostly going to be used for high performance systems, like games or programs written for a lower level. Since this is something we can also use with Typed Array's it will help save memory since we are not creating new space in memory, we are simply moving it.