Lesson | Wednesday

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / ES6 Maps and Sets

Text

In this lesson, we'll cover two more useful features of ES6: **Maps** and **Sets**. ES6 also introduced the **WeakMap** and **WeakSet** but these two data structures are more obscure and not as widely useful so we won't cover them here. You won't be required to utilize either Maps or Sets for this section's independent project but you are encouraged to explore them in class.

Maps

We'll often want to map specific keys to values. In fact, we've done this quite a bit so far with objects. For instance, let's say we're making a game where we're exploring a dungeon. Each room should have a specific key corresponding to a door number. The value associated with each key should be a description of the room.

Here's an example using a basic object:

```
const dungeon = {
  1: "The room is dark and has no windows.",
  2: "There are spiderwebs everywhere.",
  3: "There is a book on a stone pedestal. The book appears
to be glowing."
};
```

This object is being used to map specific keys to specific values. If a user enters door #3, for example, we'd want them to get the corresponding description `"There is a book on a stone pedestal. The book appears to be glowing."`.

In this situation, we could use a Map instead of an object literal. A Map is just a special kind of object that also maps keys to values. There are several similarities between a Map and a basic object but a Map has a few key advantages:

- The insertion order of key-values is preserved in a Map. This isn't the case for a basic object.
- Maps have a `size` property so we can easily see how many key-value pairs they're holding. A basic object doesn't have this functionality.
- Maps have convenient utility methods such as `Map.prototype.clear()` which basic objects don't have.
- Maps are iterable while basic objects are not. With a basic object, we need to first grab the keys and use these to iterate through the object.
- Maps can have any data type as a key while objects can only have strings or symbols as keys.

For this last one, you may be wondering how we were able to create the basic `dungeon` object above. Doesn't it have integers as keys? Actually, no. Let's check out the type of the first object key in the DevTools console:

```
> typeof Object.keys(dungeon)[0];
String
```

When we do this, we'll see that the key is actually a string, not an integer.

At this point, it should be clear that there are quite a few benefits to using Maps.

So how can we implement one using the dungeon example above?

Well, we can instantiate a map with values like this:

```
let dungeon = new Map(
  [
    [1, "The room is dark and has no windows."],
    [2, "There are spiderwebs everywhere."],
    [3, "There is a book on a stone pedestal. The book appe
ars to be glowing."]
  ]
);
```

As we can see, we instantiate key values inside a map by creating an array of arrays. The outer array holds all the key-value pairs. Each inner array holds a single key-value pair separated by a comma. The key is the first element of the array and the value is the second.

If we want to just instantiate an empty map, we can do that, too:

```
let map = new Map();
```

Now let's return to the dungeon we're building. Here's how we can add another room to our Map:

```
dungeon.set(4, "The room is full of sleeping bats.");
```

We use the `Map.prototype.set()` method to add key-value pairs to a map. The method takes two arguments: the first is the key, the second is the value.

To delete a room, we just need to specify the key:

```
dungeon.delete(4);
```

We can check to see if the dungeon has a specific room:

```
dungeon.has(12);
```

This will return `false`.

We can also get the value associated with a key:

```
dungeon.get(3);
```

This will return the associated value, which is:

```
"There is a book on a stone pedestal. The book appears to b
e glowing."
```

Note that if a value doesn't exist in a Map, the return will be `undefined`. Sometimes it will be a good idea to call `Map.prototype.has()` before looking for a value just in case.

Finally, we can iterate directly over a Map, unlike with an object. For instance, there's `Map.prototype.forEach()`:

```
dungeon.forEach(function(value, key) {
  console.log(value, key);
});
```

Note that the first argument corresponds to the values in the Map while the second (optional) argument corresponds to the keys. We could rewrite the above with different parameters:

```
dungeon.forEach(function(description, roomNumber) {
  console.log(description, roomNumber);
});
```

The result will be the same because the first argument still relates to values while the second relates to keys.

Check out the Mozilla documentation on Map (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) for more information. Here are a few things from the documentation you might want to look over:

- Iterating over a Map with a `for...of` loop.
- The `Map.prototype.keys()`, `Map.prototype.values()`, and `Map.prototype.entries()` methods, which are all iterable and specifically allow you to iterate over keys, values, and entries (which return both keys and values) respectively.

You won't be expected to use Maps on this section's independent project. However, we still recommend practicing with them and using them in your code when applicable.

Sets

Sometimes you'll want every element in a collection to be unique. While it's possible to create an array and then check that array for duplicates every time a new value is inserted, it's not necessary to do so. Instead, we can use a Set to enforce uniqueness for us. However, it's important to be aware that Sets don't really behave like arrays even though they are collections for holding elements.

We can create an empty Set like this:

```
let set = new Set();
```

We could also instantiate a Set with existing values. For instance:

```
let numbers = new Set([4, 9, 12, 4, 7]);
```

Note that there is a duplicate above. However, once we create the Set and check the value of `numbers`, we'll see that the duplicate has been removed! If you ever have an array of things and you want to remove duplicates, you can just save it as a Set. By the way, you can also pass a string in as an argument to a set — it will automatically be broken up into letters and all duplicate letters will be removed.

Here's how we can add values to a Set:

```
> numbers.add(32);
Set(5) {4, 9, 12, 7, 32}
```

What happens if we try to add a value that already exists in the set?

```
> numbers.add(4);
Set(5) {4, 9, 12, 7, 32}
```

As you can see, nothing at all. `Set.prototype.add()` just returns the full Set. There's no message if you try to add a duplicate. A Set just quietly enforces uniqueness.

We can also remove values from a Set:

```
> numbers.delete(4);
true
```

Note that this returns a boolean based on whether the value was successfully deleted. If it were to return `false`, that would just mean that the value wasn't in the Set in the first place.

We can also check to see if a Set has a value:

```
> numbers.has(9);
true
```

```
> numbers.has(51);
false
```

One very important thing to consider about Sets is that we can't grab a value by its index. For instance, we can't use bracket notation:

```
// Won't work!
numbers[0];
```

If getting an element by its index is necessary for your use case, a Set isn't the way to go. Stick with an array.

Like a Map, a Set is iterable. We can use `Set.prototype.forEach()` or a `for...of` loop to iterate through a Set (same as we can with a Map).

Here's an example:

```
numbers.forEach(function(number) {
  console.log(number);
});
```

For more information, see the Mozilla documentation on Set (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set). You will not be expected to utilize Sets in an independent project. However, you are encouraged to explore them in your own code.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/es6-array-and-object-destructuring)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/haiku-creator-rpg-sudoku-solver-two-day-project-part-2)

(http://www.epicodus.com)