Iterables & Iterators

Learn what an iterable is and what it does. Learn how the spread operator and for-of loops work under the hood and how to take advantage of them.

ES2015 introduced the **iterable** protocol. This is a way for objects to describe how they should behave when under iteration, or when we are trying to access their elements.

JavaScript Iterables

In JavaScript, an iterable is an object that has the following qualities:

- Has a property method, the key for which is `Symbol.iterator`. The method should:
    - Return an **iterator**. An iterator is an object with a `next` method. An iterator's `next` method should return an object that has the following properties:
        - `value`, any type
        - `done`, a Boolean

Let's build up an iterable using these rules.

Creating an Iterable

An iterable is an object with a method, the key for which is `Symbol.iterator`:

```
const obj = {
    [Symbol.iterator]() {}
};
```

The method should return an **iterator**. An iterator is an object with a `next` method that follows certain rules.

```
const obj = {
    [Symbol.iterator]() {
            return {
                next() {}
        };
    }
};
```

The `next` method of an iterator should return an object with `value` (any type) and `done` (Boolean).

```
const obj = {
    [Symbol.iterator]() {
            return {
                next() {
                    return {
                        value: 'some value',
                    done: false
                };
            }
        };
    }
};
```

Here, we have the template to base an iterable off of. Here's an example of one with some added functionality.

This iterator returns `val` as the value after increasing it by one, and `done` is `true` when that value is greater than 10.

```
const obj = {
    [Symbol.iterator]() {
        let val = 0;

        return {
            next() {
                return { value: val++, done: val > obj.maxValue };
            }
        };
    },

    maxValue: 10
};
```

That's it. `obj` above is an iterable. What's so special about this?

What Iterables Do

By following this format, we've created something very powerful. Observe the following.

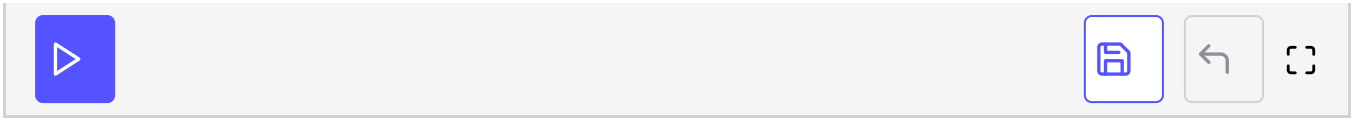
```
const obj = {
    [Symbol.iterator]() {
        let val = 0;

        return {
            next() {
                return { value: val++, done: val > obj.maxValue };
            }
        };
    },

    maxValue: 10
};

for(const value of obj) {
    console.log(value);
}
/* -> 0
   -> 1
   -> ...
   -> 9
*/

console.log(...obj);
// -> 0 1 2 3 4 5 6 7 8 9
```



for-of and Spread

By following the iterable format, we've defined how our object behaves inside for-of loops and with the spread operator. This is because for-of loops and the spread operator use iterables under the hood.

Arrays have their own default iterables built in. This is what the spread operator uses and why it works as it does.

`value` & `done`

For-of loops and the spread operator will continue spitting out values called by the `next()` method until `done` is equal to `true`. This is our **stop condition**, comparable to a recursive function's stop condition.

The `value` property is what will actually be assigned in our loops and spread using the spread operator.

Manual Iteration

Custom Iterables

We can iterate manually by using the object returned by our `Symbol.iterator` method. This returned object is an iterator.

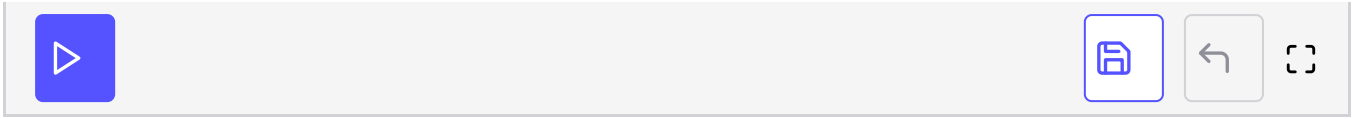
```
const obj = {
    [Symbol.iterator]() {
        let val = 0;

        return {
            next() {
                return { value: val++, done: val > obj.maxValue };
            }
        };
    },

    maxValue: 10
};

const iterator = obj[Symbol.iterator]();

console.log(iterator.next().value); // -> 0
console.log(iterator.next().value); // -> 1
console.log(iterator.next().value); // -> 2
console.log(iterator.next().value); // -> 3
```

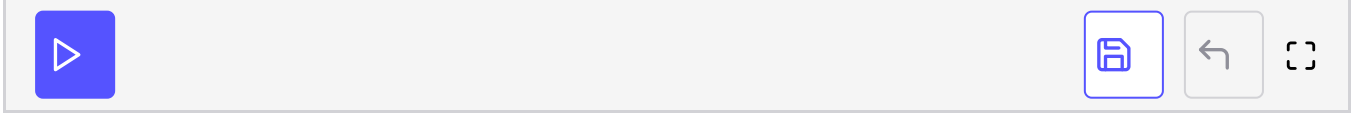


Built-In Iterators

We can do this with built-in iterators as well. This will work with arrays, strings, maps, and sets, as all are iterables.

```
const arr = [1, 2, 3, 4, 5];
const iterator = arr[Symbol.iterator]();

console.log(iterator.next().value); // -> 1
console.log(iterator.next().value); // -> 2
console.log(iterator.next().value); // -> 3
```



Iterators can return values indefinitely. All that's required is that `done` always

equals `false`.

Note that attempting to spread an iterable with an infinite iterator will result in an infinite loop and break our program.

In a for-of loop, we'll have to write our own stop condition to break out of the loop.

Iterables are very important when used through generators, which we'll cover in the next lesson.

That's it for iterables & iterators.