Iterables and Iterators

introduction to Iterables and Iterators and their behavior in Javascript

Now that we have introduced the `for...of` loop and symbols, we can comfortably talk about iterators.

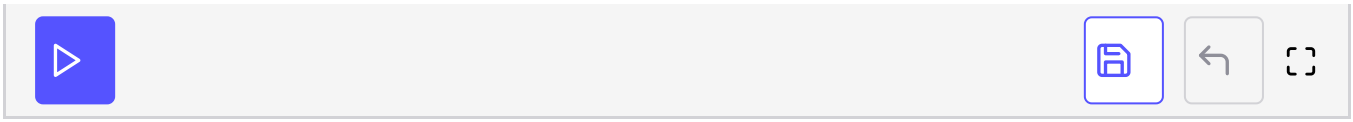
It is worth for you to learn about iterators, especially if you like lazy evaluation, or you want to be able to describe infinite sequences. Understanding of iterators also helps you understand generators, promises, sets, and maps better.

Once we cover the fundamentals of iterators, we will use our knowledge to understand how generators work.

ES6 comes with the *iterable* protocol. The protocol defines the iterating behavior of JavaScript objects.

An *iterable object* has an iterator method with the key `Symbol.iterator`. This method returns an *iterator object*.

```
let iterableObject = {
  [Symbol.iterator]() { return iteratorObject; }
};
```

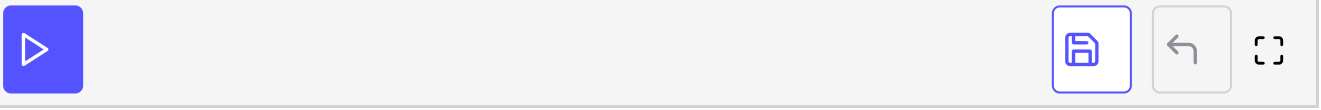


`Symbol.iterator` is a *well-known symbol*. We will now use `Symbol.iterator` to describe an iterable object. Note that we are using this construct for the sake of understanding how iterators work. Technically, you will hardly ever need `Symbol.iterator` in your code. You will soon learn another way to define iterables.

An *iterator object* is a data structure that has a `next` method. When calling this method on the iterator, it returns the next element, and a boolean

signaling whether we reached the end of the iteration.

```
// Place this before iterableObject
let iteratorObject = {
    next() {
        return {
            done: true,
            value: null
        };
    }
};
```



The return value of the `next` function has two keys:

- `done` is treated as a boolean. When `done` is true, the iteration ends, and `value` is not considered in the iteration.
- `value` is the upcoming value of the iteration. It is considered in the iteration if and only if `done` is false. When `done` is true, `value` becomes the return value of the iterator.

Let's create a countdown object as an example:

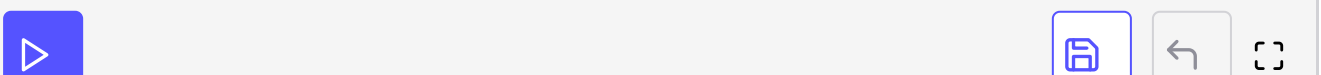
```
let countdownIterator = {
    countdown: 10,
    next() {
        this.countdown -= 1;
        return {
            done: this.countdown === 0,
            value: this.countdown
        };
    }
};

let countdownIterable = {
    [Symbol.iterator]() {
        return Object.assign( {}, countdownIterator )
    }
};

let iterator = countdownIterable[Symbol.iterator]();

console.log(iterator.next());
//> Object {done: false, value: 9}

console.log(iterator.next());
//> Object {done: false, value: 8}
```



Note that the state of the iteration is preserved.

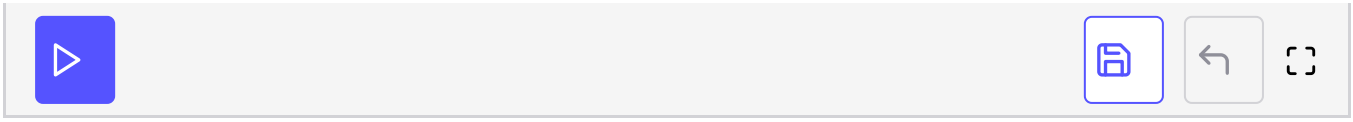
The role of `Object.assign` is that we create a shallow copy of the iterator object each time the iterable returns an iterator. This allows us to have multiple iterators on the same iterable object, storing their own internal state. Without `Object.assign`, we would just have multiple references to the same iterator object. Consider the following code:

```
let secondIterator = countdownIterable[Symbol.iterator]();
let thirdIterator = countdownIterable[Symbol.iterator]();

console.log( secondIterator.next() );
//> Object {done: false, value: 9}

console.log( thirdIterator.next() );
//> Object {done: false, value: 9}

console.log( secondIterator.next() );
//> Object {done: false, value: 8}
```



We will now learn how to make use of iterators and iterable objects.