Generators and Iterators

generators returning Iterators that are also Iterables

Recall our string iterator example to refresh what iterable objects and iterators are:

```
let message = 'ok';
let stringIterator = message[Symbol.iterator]();
```

We call the `next` method of `stringIterator` to get the next element:

```
console.log( stringIterator.next() );
```

However, before we learned about iterators, we used the *iterable object* in a `for-of` loop, not the iterator:

```
let message = 'ok';
for ( let ch of message ) {
    console.log( ch );
}
```

Let's summarize what *iterables* and *iterators* are once more:

- Iterable objects have a `[Symbol.iterator]` method that returns an iterator.
- Iterator objects have a `next` method that returns an object with keys `value` and `done`

`value` and `done`.

Generator functions return an object that is both an iterable and an iterator. Generator functions have:

- a `[Symbol.iterator]` method to return their iterator,
- a `next` method to perform the iteration.

As a consequence, the return value of generator functions can be used in `for-of` loops, after the spread operator, and in all places where iterables are consumed.

```
function *getLampIterator() {
    yield 'red';
    yield 'green';
    return 'lastValue';
    // implicit: return undefined;
}

let lampIterator = getLampIterator();

console.log( lampIterator.next() );
//> {value: 'red', done: false}

console.log( [...lampIterator] );
//> ['green']
```

In the above example, `[...lampIterator]` contains the remaining values of the iteration in an array.

Now, let’s move on to Iterators and destructuring.