Exercise on Iterators and Generators

We will play around with Iterators and Generators to get a deeper understanding of how they work.

These exercises help you explore how iterators and generators work. You will get a chance to play around with iterators and generators, which will result in a higher depth of learning experience for you than reading about the edge cases.

You can also find out if you already know enough to command these edge cases without learning more about iterators and generators.

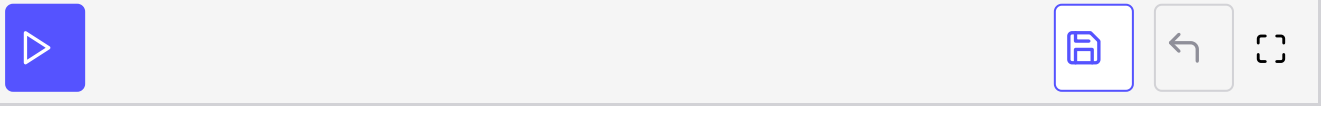
Exercise 1:

What happens if we use a string iterator instead of an iterable object in a `for-of` loop?

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

messageIterator.next();

for ( let item of messageIterator ) {
    console.log( item );
}
```

Solution

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

messageIterator.next();

for ( let item of messageIterator ) {
    console.log( item );
}
```



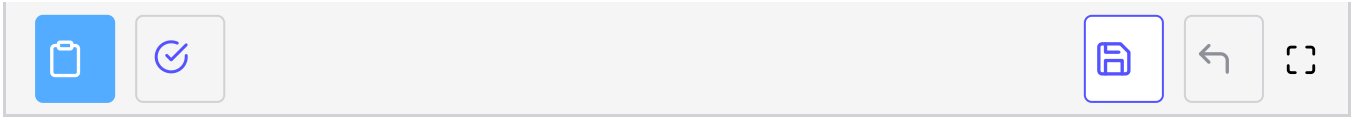
Similarly to generators, in case of strings, arrays, DOM elements, sets, and maps, an iterator object is also an iterable. Therefore, in the `for-of` loop, the remaining k letter is printed out.

Exercise 2:

Create a countdown iterator that counts from 9 to 1. Use generator functions!

```
let getCountdownIterator = // Your code comes here

console.log( [ ...getCountdownIterator() ] );
> [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
let getCountdownIterator = function *() {
  //Write your Code here
}
```

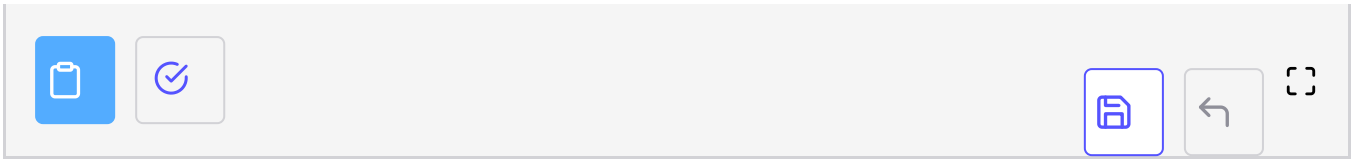


Exercise 3:

Make the following object iterable:

```
let todoList = {
    todoItems: [],
    addItem( description ) {
        this.todoItems.push( { description, done: false } );
        return this;
    },
    crossOutItem( index ) {
        if ( index < this.todoItems.length ) {
            this.todoItems[index].done = true;
        }
        return this;
    }
};

todoList.addItem( 'task 1' ).addItem( 'task 2' ).crossOutItem( 0 );

let iterableTodoList = /// ???;

for ( let item of iterableTodoList ) {
    console.log( item );
}

// Without your code, you get the following error:
```

```
// TypeError: iterableTodoList is not iterable
```



We could use well known symbols to make todoList iterable. We can add a `*[Symbol.iterator]` generator function that yields the elements of the array. This will make the `todoList` object iterable, yielding the elements of `todoItems` one by one.

```
let todoList = {
todoItems: [],
*[Symbol.iterator]() {
yield* this.todoItems;
}
addItem( description ) {
this.todoItems.push( { description, done: false } );
return this;
},
crossOutItem( index ) {
if ( index < this.todoItems.length ) {
this.todoItems[index].done = true;
}
return this;
}
};
let iterableTodoList = todoList;
```

This solution is not compatible with the code of the exercise. Furthermore, we prefer staying away from well-known symbols. This solution reads a bit like a hack.

So make the code more semantic by following the patterns of the exercise. The solution given for this exercise looks cleaner even if you have to type more characters.

Exercise 4:

Determine the values logged to the console without running the code. Instead of just writing down the values, formulate your thought process and explain to yourself how the code runs line by line. You should execute each statement, loop, and declaration one by one.

```
let errorDemo = function *() {
    yield 1;
    yeild 'Error yielding the next result';
    yield 2;

}

let it = errorDemo();

// Execute one statement at a time to avoid
// skipping lines after the first thrown error.

console.log( it.next() );

console.log( it.next() );

console.log( [...errorDemo()] );

for ( let element of errorDemo() ) {
    console.log( element );
}
```

Solution

```
let errorDemo = function *() {
    yield 1;
    yield 'Error yielding the next result';
    yield 2;
}

let it = errorDemo();

// Execute one statement at a time to avoid
// skipping lines after the first thrown error.

console.log( it.next() );

console.log( it.next() );

console.log( [...errorDemo()] );

for ( let element of errorDemo() ) {
    console.log( element );
}
```

We created three iterables in total: it, one in the statement in the spread operator, and one in the for-of loop.

In the example with the next calls, the second call results in a thrown error. In the spread operator example, the expression cannot be evaluated, because an error is thrown.

In the for-of example, the first element is printed out, then the error stopped the execution of the loop.

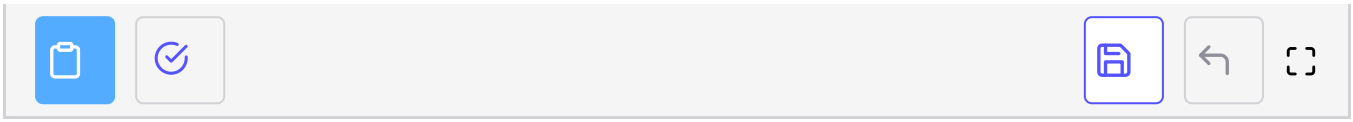
Exercise 5:

Create an infinite sequence that generates the next value of the Fibonacci sequence.

The Fibonacci sequence is defined as follows:

- `fib( 0 ) = 0`
- `fib( 1 ) = 1`
- for `n > 1`, `fib( n ) = fib( n - 1 ) + fib( n - 2 )`

```
function *fibonacci() {
  //Write your code here
}
```



Note that you only want to get the `next()` element of an infinite sequence. Executing `[...fibonacci()]` will skyrocket your CPU usage, speed up your CPU fan, and then crash your browser.

Exercise 6:

Create a lazy `filter` generator function. Filter the elements of the Fibonacci sequence by keeping the even values only.

```
function *filter( iterable, filterFunction ) {
    // insert code here
}
```

```
//Assume Fibonacci function already defined

function *filter( iterable, filterFunction ) {
    // Write your code here
}
```