JavaScript Generators

Summary: in this tutorial, you will learn about JavaScript Generators and how to use them effectively.

Introduction to JavaScript Generators

In JavaScript, a regular function is executed based on the run-to-completion model. It cannot pause midway and then continues from where it paused. For example:

```
function foo() {
    console.log('I');
    console.log('cannot');
    console.log('pause');
}
```

The `foo()` function executes from top to bottom. The only way to exit the `foo()` is by returning from it or throwing an error. If you invoke the `foo()` function again, it will start the execution from the top to bottom.

```
foo();
```

Output:

```
I
cannot
pause
```

ES6 introduces a new kind of function that is different from a regular function: function generator or generator.

A generator can pause midway and then continues from where it paused. For example:

```
function* generate() {
    console.log('invoked 1st time');
    yield 1;
    console.log('invoked 2nd time');
    yield 2;
}
```

Let's examine the `generate()` function in detail.

- First, you see the asterisk (`*`) after the `function` keyword. The asterisk denotes that the `generate()` is a generator, not a normal function.
- Second, the `yield` statement returns a value and pauses the execution of the function.

The following code invokes the `generate()` generator:

```
let gen = generate();
```

When you invoke the `generate()` generator:

- First, you see nothing in the console. If the `generate()` were a regular function, you would expect to see some messages.
- Second, you get something back from `generate()` as a returned value.

Let's show the returned value on the console:

```
console.log(gen);
```

Output:

```
Object [Generator] {}
```

So, a generator returns a `Generator` object without executing its body when it is invoked.

The `Generator` object returns another object with two properties: `done` and `value`. In other words, a `Generator` object is iterable.

The following calls the `next()` method on the `Generator` object:

```
let result = gen.next();
console.log(result);
```

Output:

```
invoked 1st time
{ value: 1, done: false }
```

As you can see, the Generator object executes its body which outputs message `'invoked 1st time'` at line 1 and returns the value 1 at line 2.

The `yield` statement returns 1 and pauses the generator at line 2.

Similarly, the following code invokes the `next()` method of the Generator second time:

```
result = gen.next();
console.log(result);
```

Output:

```
invoked 2nd time
{ value: 2, done: false }
```

This time the Generator resumes its execution from line 3 that outputs the message `'invoked 2nd time'` and returns (or yield) 2.

The following invokes the `next()` method of the generator object a third time:

```
result = gen.next();
console.log(result);
```

Output:

```
{ value: undefined, done: true }
```

Since a generator is iterable, you can use the `for...of` loop:

```
for (const g of gen) {
    console.log(g);
}
```

Here is the output:

```
invoked 1st time
1
invoked 2nd time
2
```

More JavaScript generator examples

The following example illustrates how to use a generator to generate a never-ending sequence:

```
function* forever() {
    let index = 0;
    while (true) {
        yield index++;
    }
}

let f = forever();
console.log(f.next()); // 0
console.log(f.next()); // 1
console.log(f.next()); // 2
```

Each time you call the `next()` method of the `forever` generator, it returns the next number in the sequence starting from 0.

Using generators to implement iterators

When you implement an iterator, you have to manually define the `next()` method. In the `next()` method, you also have to manually save the state of the current element.

Since generators are iterables, they can help you simplify the code for implementing iterator.

The following is a `Sequence` iterator created in the iterator tutorial:

```
class Sequence {
    constructor( start = 0, end = Infinity, interval = 1 ) {
        this.start = start;
        this.end = end;
        this.interval = interval;
    }
    [Symbol.iterator]() {
        let counter = 0;
        let nextIndex = this.start;
        return  {
            next: () => {
                if ( nextIndex < this.end ) {
                    let result = { value: nextIndex,  done: false }
                    nextIndex += this.interval;
                    counter++;
                    return result;
                }
                return { value: counter, done: true };
            }
        }
    }
}
```

And here is the new Sequence iterator that uses a generator:

```
class Sequence {
    constructor( start = 0, end = Infinity, interval = 1 ) {
        this.start = start;
        this.end = end;
```

```
        this.interval = interval;
    }
    * [Symbol.iterator]() {
        for( let index = this.start; index <= this.end; index += this.interval ) {
            yield index;
        }
    }
}
```

As you can see, the method Symbol.iterator is much simpler by using the generator.

The following script uses the Sequence iterator to generate a sequence of odd numbers from 1 to 10:

```
let oddNumbers = new Sequence(1, 10, 2);

for (const num of oddNumbers) {
    console.log(num);
}
```

Output:

```
1
3
5
7
9
```

Using a generator to implement the Bag data structure

A Bag is a data structure that has the ability to collect elements and iterate through elements. It doesn't support removing items.

The following script implements the `Bag` data structure:

```
class Bag {
    constructor() {
        this.elements = [];
    }
    isEmpty() {
        return this.elements.length === 0;
    }
    add(element) {
        this.elements.push(element);
    }
    * [Symbol.iterator]() {
        for (let element of this.elements) {
            yield element;
        }
    }
}

let bag = new Bag();

bag.add(1);
bag.add(2);
bag.add(3);

for (let e of bag) {
    console.log(e);
}
```

Output:

```
1
2
3
```

Summary

- Generators are created by the generator function `function* f(){}`.
- Generators do not execute its body immediately when they are invoked.

- Generators can pause midway and resumes their executions where they were paused. The `yield` statement pauses the execution of a generator and returns a value.
- Generators are iterable so you can use them with the `for...of` loop.