Generators

Learn about ES2015 generators, a new type of function. Learn how to pause a function and dynamically change its state. These allow us to get multiple values out of a function.

Generators are functions that maintain their own internal state. A generator works as a factory for iteratables. They return a Generator object, which is both an iterable and an iterator. That is, they can be iterated using for-of loops and the spread operator, and they can be iterated through manually using `next`.

Generators

Creating a Generator

A generator is created by declaring a function with a special character: `*`.

```
function* generator() {}
```

`yield`

Instead of using `return` statements, generators use a new keyword, `yield`. We can yield values from a generator multiple times. The values we yield are used as if they were coming from an iterable.

```
function* generator() {
    yield 1;
    yield 2;
    yield 3;
}

const iterable = generator();
console.log(...iterable); // -> 1 2 3

const iterator2 = generator();
console.log(iterator2.next()); // -> { value: 1, done: false }
console.log(iterator2.next()); // -> { value: 2, done: false }
console.log(iterator2.next()); // -> { value: 3, done: false }
console.log(iterator2.next()); // -> { value: undefined, done: true }
```

Note that we created two iterators above. That is because getting the values out of one “uses up” its values. The function maintains state and once something is yielded, it moves on, permanently.

Function Pausing

The yield keyword allows us to pause execution of a function. When we use the iterator shown above, it will run our function until the first `yield` statement. It will then pause execution and provide the value specified. When we call `next` again, it will continue running right where it left off and again pause at the next `yield` statement.

```
function* generator() {
    console.log('First yield');
    yield;

    console.log('Second yield');
    yield;

    console.log('Third yield');
    yield;
}

const iterator = generator();
iterator.next(); // -> First yield
iterator.next(); // -> Second yield
iterator.next(); // -> Third yield
```

Returning from a Generator

We can use return statements in iterators. When we return from an iterator, it ends the iterator. `done` changes to `true` and further `yield` and `return` statements do nothing.

```
function* generator() {
    yield 1;
    yield 2;
    return 3;
    yield 4;
}

const iterator = generator();
console.log(iterator.next()); // -> { value: 1, done: false }
console.log(iterator.next()); // -> { value: 2, done: false }
console.log(iterator.next()); // -> { value: 3, done: true }
console.log(iterator.next()); // -> { value: undefined, done: true }
```

Yielding Into Generators

Generators allow us to dynamically pass data into them.

```
function* generator() {
    yield 1;
    yield 2;

    let receivedValue = yield 3;

    console.log('Received:', receivedValue);
    yield 4;
}

const iterator = generator();
console.log(iterator.next()); // -> { value: 1, done: false }
console.log(iterator.next()); // -> { value: 2, done: false }
console.log(iterator.next()); // -> { value: 3, done: false }
console.log(iterator.next(17));
// -> Received: 17
// -> { value: 4, done: false }
```

We can pass a parameter into our `next()` call. This will be received inside the generator function and can be stored in a variable for use.

In the function above, `yield 3` on line 5 gets dynamically replaced by the value we provide on line 15.

This allows us to manage a generator's internal state dynamically.

```
function* counter() {
    let count = 0;

    while(true) {
        const reset = yield count++;

        if(reset) {
            count = 0;
        }
    }
}

const iterable = counter();
const iterable = counter();
console.log(iterable.next()); // -> 0
console.log(iterable.next()); // -> 1
console.log(iterable.next()); // -> 2

console.log(iterable.next(true)); // -> 0
console.log(iterable.next()); // -> 1
console.log(iterable.next()); // -> 2
```

Here, we've made an infinitely incrementing counter that we can reset anytime we want.

`yield` with `*`

We can use `yield` with the `*` character for some advanced behavior. Using `yield*` with an iterator will yield the values of that iterator one by one.

```
function* generator() {
    yield* ['abc', 'def', 'ghi'];
}

const iterator = generator();
console.log(iterator.next().value); // -> abc
console.log(iterator.next().value); // -> def
console.log(iterator.next().value); // -> ghi
```

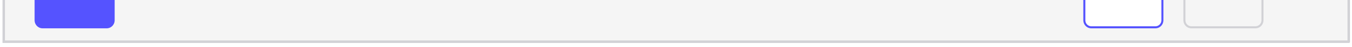
This allows us to chain generators.

```
function* generator2(i) {
    const maxValue = i + 5;

    while(i < maxValue) {
        yield i++;
    }
}

function* generator() {
    yield* generator2(0);
    yield* generator2(100);
}

console.log(...generator());
// -> 0 1 2 3 4 100 101 102 103 104
```



That’s it for generators.