



🏠 → [The JavaScript language](#) → [Generators, advanced iteration](#)

📅 7th February 2020

Generators

Regular functions return only one, single value (or nothing).

Generators can return (“yield”) multiple values, one after another, on-demand. They work great with [iterables](#), allowing to create data streams with ease.

Generator functions

To create a generator, we need a special syntax construct: `function*`, so-called “generator function”.

It looks like this:

```
1 function* generateSequence() {  
2   yield 1;  
3   yield 2;  
4   return 3;  
5 }
```

Generator functions behave differently from regular ones. When such function is called, it doesn't run its code. Instead it returns a special object, called “generator object”, to manage the execution.

Here, take a look:

```
1 function* generateSequence() {  
2   yield 1;  
3   yield 2;  
4   return 3;  
5 }  
6  
7 // "generator function" creates "generator object"  
8 let generator = generateSequence();  
9 alert(generator); // [object Generator]
```



The function code execution hasn't started yet:

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



The main method of a generator is `next()`. When called, it runs the execution until the nearest `yield` `<value>` statement (`value` can be omitted, then it's `undefined`). Then the function execution pauses, and the yielded `value` is returned to the outer code.

The result of `next()` is always an object with two properties:

- `value` : the yielded value.
- `done` : `true` if the function code has finished, otherwise `false`.


For instance, here we create the generator and get its first yielded value:

```
1 function* generateSequence() {  
2   yield 1;  
3   yield 2;  
4   return 3;  
5 }  
6  
7 let generator = generateSequence();  
8  
9 let one = generator.next();  
10  
11 alert(JSON.stringify(one)); // {value: 1, done: false}
```



As of now, we got the first value only, and the function execution is on the second line:

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

 `{value: 1, done: false}`

Let's call `generator.next()` again. It resumes the code execution and returns the next `yield`:

```
1 let two = generator.next();  
2  
3 alert(JSON.stringify(two)); // {value: 2, done: false}
```


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

 `{value: 2, done: false}`

And, if we call it a third time, the execution reaches the `return` statement that finishes the function:

```
1 let three = generator.next();
2
3 alert(JSON.stringify(three)); // {value: 3, done: true}
```

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



{value: 3, done: true}

Now the generator is done. We should see it from `done: true` and process `value: 3` as the final result.

New calls to `generator.next()` don't make sense any more. If we do them, they return the same object: `{done: true}`.

i `function* f(...)` or `function *f(...)` ?

Both syntaxes are correct.

But usually the first syntax is preferred, as the star `*` denotes that it's a generator function, it describes the kind, not the name, so it should stick with the `function` keyword.

Generators are iterable

As you probably already guessed looking at the `next()` method, generators are [iterable](#).

We can loop over their values using `for..of` :

```
1 function* generateSequence() {
2   yield 1;
3   yield 2;
4   return 3;
5 }
6
7 let generator = generateSequence();
8
9 for(let value of generator) {
10  alert(value); // 1, then 2
11 }
```



Looks a lot nicer than calling `.next().value`, right?

...But please note: the example above shows `1`, then `2`, and that's all. It doesn't show `3` !

It's because `for..of` iteration ignores the last `value`, when `done: true`. So, if we want all results to be shown by `for..of`, we must return them with `yield` :

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



```
3   yield 2;
4   yield 3;
5 }
6
7 let generator = generateSequence();
8
9 for(let value of generator) {
10  alert(value); // 1, then 2, then 3
11 }
```

As generators are iterable, we can call all related functionality, e.g. the spread syntax `...`:

```
1 function* generateSequence() {
2   yield 1;
3   yield 2;
4   yield 3;
5 }
6
7 let sequence = [0, ...generateSequence()];
8
9 alert(sequence); // 0, 1, 2, 3
```

In the code above, `...generateSequence()` turns the iterable generator object into an array of items (read more about the spread syntax in the chapter [Rest parameters and spread syntax](#))

Using generators for iterables

Some time ago, in the chapter [Iterables](#) we created an iterable `range` object that returns values from `..to`.

Here, let's remember the code:

```
1 let range = {
2   from: 1,
3   to: 5,
4
5   // for..of range calls this method once in the very beginning
6   [Symbol.iterator]() {
7     // ...it returns the iterator object:
8     // onward, for..of works only with that object, asking it for next values
9     return {
10      current: this.from,
11      last: this.to,
12
13      // next() is called on each iteration by the for..of loop
14      next() {
15        // it should return the value as an object {done:..., value :...}
16        if (this.current <= this.last) {
17          return { done: false, value: this.current++ };
18        } else {
19          return { done: true };
20        }
21      }
22    };
23  }
24 };
```

```
25
26 // iteration over range returns numbers from range.from to range.to
27 alert([...range]); // 1,2,3,4,5
```

We can use a generator function for iteration by providing it as `Symbol.iterator`.

Here's the same `range`, but much more compact:

```
1  let range = {
2    from: 1,
3    to: 5,
4
5    *[Symbol.iterator]() { // a shorthand for [Symbol.iterator]: function*()
6      for(let value = this.from; value <= this.to; value++) {
7        yield value;
8      }
9    }
10 };
11
12 alert( [...range] ); // 1,2,3,4,5
```

That works, because `range[Symbol.iterator]()` now returns a generator, and generator methods are exactly what `for...of` expects:

- it has a `.next()` method
- that returns values in the form `{value: ..., done: true/false}`

That's not a coincidence, of course. Generators were added to JavaScript language with iterators in mind, to implement them easily.

The variant with a generator is much more concise than the original iterable code of `range`, and keeps the same functionality.

i Generators may generate values forever

In the examples above we generated finite sequences, but we can also make a generator that yields values forever. For instance, an unending sequence of pseudo-random numbers.

That surely would require a `break` (or `return`) in `for...of` over such generator. Otherwise, the loop would repeat forever and hang.

Generator composition

Generator composition is a special feature of generators that allows to transparently “embed” generators in each other.

For instance, we have a function that generates a sequence of numbers:

```
1  function* generateSequence(start, end) {
2    for (let i = start; i <= end; i++) yield i;
3  }
```

Now we'd like to reuse it to generate a more complex sequence:

- first, digits 0..9 (with character codes 48...57),
- followed by uppercase alphabet letters A..Z (character codes 65...90)
- followed by lowercase alphabet letters a..z (character codes 97...122)

We can use this sequence e.g. to create passwords by selecting characters from it (could add syntax characters as well), but let's generate it first.

In a regular function, to combine results from multiple other functions, we call them, store the results, and then join at the end.

For generators, there's a special `yield*` syntax to "embed" (compose) one generator into another.

The composed generator:

```
1 function* generateSequence(start, end) {  
2   for (let i = start; i <= end; i++) yield i;  
3 }  
4  
5 function* generatePasswordCodes() {  
6  
7   // 0..9  
8   yield* generateSequence(48, 57);  
9  
10  // A..Z  
11  yield* generateSequence(65, 90);  
12  
13  // a..z  
14  yield* generateSequence(97, 122);  
15  
16 }  
17  
18 let str = '';  
19  
20 for(let code of generatePasswordCodes()) {  
21   str += String.fromCharCode(code);  
22 }  
23  
24 alert(str); // 0..9A..Za..z
```

The `yield*` directive *delegates* the execution to another generator. This term means that `yield* gen` iterates over the generator `gen` and transparently forwards its yields outside. As if the values were yielded by the outer generator.

The result is the same as if we inlined the code from nested generators:

```
1 function* generateSequence(start, end) {  
2   for (let i = start; i <= end; i++) yield i;  
3 }  
4  
5 function* generateAlphaNum() {  
6  
7   // yield* generateSequence(48, 57);  
8   for (let i = 48; i <= 57; i++) yield i;  
9
```

```

10 // yield* generateSequence(65, 90);
11 for (let i = 65; i <= 90; i++) yield i;
12
13 // yield* generateSequence(97, 122);
14 for (let i = 97; i <= 122; i++) yield i;
15
16 }
17
18 let str = '';
19
20 for(let code of generateAlphaNum()) {
21   str += String.fromCharCode(code);
22 }
23
24 alert(str); // 0..9A..Za..z

```

A generator composition is a natural way to insert a flow of one generator into another. It doesn't use extra memory to store intermediate results.

“yield” is a two-way street

Until this moment, generators were similar to iterable objects, with a special syntax to generate values. But in fact they are much more powerful and flexible.

That's because `yield` is a two-way street: it not only returns the result to the outside, but also can pass the value inside the generator.

To do so, we should call `generator.next(arg)`, with an argument. That argument becomes the result of `yield`.

Let's see an example:

```

1 function* gen() {
2   // Pass a question to the outer code and wait for an answer
3   let result = yield "2 + 2 = ?"; // (*)
4
5   alert(result);
6 }
7
8 let generator = gen();
9
10 let question = generator.next().value; // <-- yield returns the value
11
12 generator.next(4); // --> pass the result into the generator

```

Generator

Calling code

```

function* gen() {
  let result = yield "2 + 2 = ?";
  alert(result); // 4
}

```

question = "2 + 2 = ?"

.next(4)

1. The first call `generator.next()` should be always made without an argument (the argument is ignored if passed). It starts the execution and returns the result of the first `yield` `"2+2=?"`. At this point the generator pauses the execution, while staying on the line `(*)`.
2. Then, as shown at the picture above, the result of `yield` gets into the `question` variable in the calling code.
3. On `generator.next(4)`, the generator resumes, and `4` gets in as the result: `let result = 4`.

Please note, the outer code does not have to immediately call `next(4)`. It may take time. That's not a problem: the generator will wait.

For instance:

```
1 // resume the generator after some time
2 setTimeout(() => generator.next(4), 1000);
```

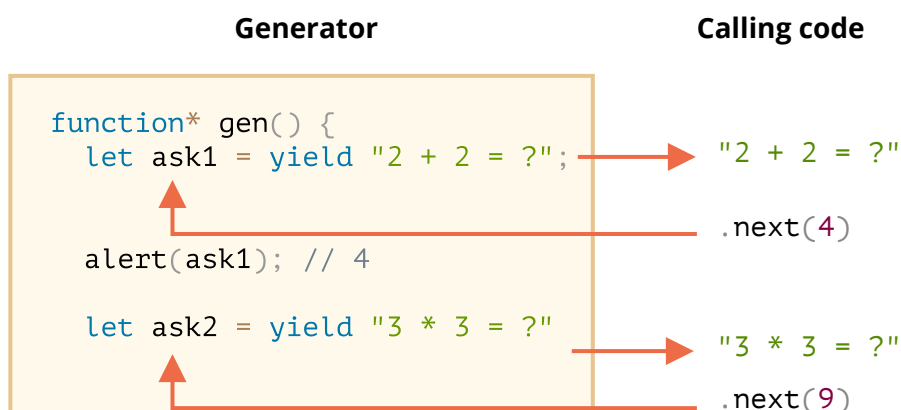
As we can see, unlike regular functions, a generator and the calling code can exchange results by passing values in `next/yield`.

To make things more obvious, here's another example, with more calls:

```
1 function* gen() {
2   let ask1 = yield "2 + 2 = ?";
3
4   alert(ask1); // 4
5
6   let ask2 = yield "3 * 3 = ?"
7
8   alert(ask2); // 9
9 }
10
11 let generator = gen();
12
13 alert( generator.next().value ); // "2 + 2 = ?"
14
15 alert( generator.next(4).value ); // "3 * 3 = ?"
16
17 alert( generator.next(9).done ); // true
```



The execution picture:




```
    alert(ask2); // 9
  }
```

1. The first `.next()` starts the execution... It reaches the first `yield`.
2. The result is returned to the outer code.
3. The second `.next(4)` passes `4` back to the generator as the result of the first `yield`, and resumes the execution.
4. ...It reaches the second `yield`, that becomes the result of the generator call.
5. The third `next(9)` passes `9` into the generator as the result of the second `yield` and resumes the execution that reaches the end of the function, so `done: true`.

It's like a "ping-pong" game. Each `next(value)` (excluding the first one) passes a value into the generator, that becomes the result of the current `yield`, and then gets back the result of the next `yield`.

generator.throw

As we observed in the examples above, the outer code may pass a value into the generator, as the result of `yield`.

...But it can also initiate (throw) an error there. That's natural, as an error is a kind of result.

To pass an error into a `yield`, we should call `generator.throw(err)`. In that case, the `err` is thrown in the line with that `yield`.

For instance, here the `yield` of `"2 + 2 = ?"` leads to an error:

```
1  function* gen() {
2    try {
3      let result = yield "2 + 2 = ?"; // (1)
4
5      alert("The execution does not reach here, because the exception is thrown
6    } catch(e) {
7      alert(e); // shows the error
8    }
9  }
10
11 let generator = gen();
12
13 let question = generator.next().value;
14
15 generator.throw(new Error("The answer is not found in my database")); // (2)
```

The error, thrown into the generator at line (2) leads to an exception in line (1) with `yield`. In the example above, `try...catch` catches it and shows it.

If we don't catch it, then just like any exception, it "falls out" the generator into the calling code.

The current line of the calling code is the line with `generator.throw`, labelled as (2). So we can catch it here, like this:



```
1 function* generate() {  
2   let result = yield "2 + 2 = ?"; // Error in this line  
3 }  
4  
5 let generator = generate();  
6  
7 let question = generator.next().value;  
8  
9 try {  
10  generator.throw(new Error("The answer is not found in my database"));  
11 } catch(e) {  
12  alert(e); // shows the error  
13 }
```

If we don't catch the error there, then, as usual, it falls through to the outer calling code (if any) and, if uncaught, kills the script.

Summary

- Generators are created by generator functions `function* f(...) {...}`.
- Inside generators (only) there exists a `yield` operator.
- The outer code and the generator may exchange results via `next/yield` calls.

In modern JavaScript, generators are rarely used. But sometimes they come in handy, because the ability of a function to exchange data with the calling code during the execution is quite unique. And, surely, they are great for making iterable objects.

Also, in the next chapter we'll learn async generators, which are used to read streams of asynchronously generated data (e.g paginated fetches over a network) in `for await ... of` loops.

In web-programming we often work with streamed data, so that's another very important use case.

✓ Tasks

Pseudo-random generator

There are many areas where we need random data.

One of them is testing. We may need random data: text, numbers, etc. to test things out well.

In JavaScript, we could use `Math.random()`. But if something goes wrong, we'd like to be able to repeat the test, using exactly the same data.

For that, so called "seeded pseudo-random generators" are used. They take a "seed", the first value, and then generate the next ones using a formula so that the same seed yields the same sequence, and hence the whole flow is easily reproducible. We only need to remember the seed to repeat it.

An example of such formula, that generates somewhat uniformly distributed values:

```
1 next = previous * 16807 % 2147483647
```

If we use `1` as the seed, the values will be:

1. 16807
2. 282475249
3. 1622650073
4. ...and so on...

The task is to create a generator function `pseudoRandom(seed)` that takes `seed` and creates the generator with this formula.

Usage example:

```
1 let generator = pseudoRandom(1);
2
3 alert(generator.next().value); // 16807
4 alert(generator.next().value); // 282475249
5 alert(generator.next().value); // 1622650073
```

[Open a sandbox with tests.](#)

solution



Previous lesson

Next lesson



Share  

 [Tutorial map](#)

Comments

- If you have suggestions what to improve - please [submit a GitHub issue](#) or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox ([plnkr](#), [JSBin](#), [codepen](#)...)