ES6 introduced a new way of working with functions and iterators in the form of **Generators (or generator functions)**. A generator is a function that **can stop midway** and then continue *from where it stopped.***In short, a generator*appears* to be a function but it *behaves* like an iterator.**

**Fun Fact**: async/await is based on generators. Read more [here](https://tc39.github.io/ecmascript-asyncawait/).

Generators are intricately linked with iterators. If you don’t know about iterators, [here](https://codeburst.io/a-simple-guide-to-es6-iterators-in-javascript-with-examples-189d052c3d8e) is an article to better your understanding of them.

[A Simple Guide to ES6 Iterators in JavaScript with Examples  
A Guest Post By: Arfat Salmancodeburst.io](https://codeburst.io/a-simple-guide-to-es6-iterators-in-javascript-with-examples-189d052c3d8e)

Here’s a simple analogy to have an intuition for generators before we proceed with the technical details.

Imagine you are reading a nail-biting techno-thriller. All engrossed in the pages of the book, you barely hear your doorbell ring. It’s the pizza delivery guy. You get up to open the door. **However, before doing that, you set a bookmark at the last page you read.** You mentally save the events of the plot. Then, you go and get your pizza. Once you return back to your room, you begin the book ***from the page that you set the bookmark on****.*You don’t begin it from the first page again. In a sense, you acted as a generator function.

**Introduction**

Let’s see how we can utilise generators to solve some common problems while programming. But before that, let’s define what generators are.

**What are Generators?**

A normal function such as this one cannot be stopped *before* it finishes its task i.e its last line is executed. It follows something called [run-to-completion](https://en.wikipedia.org/wiki/Run_to_completion_scheduling)model.

function normalFunc() {  
 console.log('I')  
 console.log('cannot')  
 console.log('be')  
 console.log('stopped.')  
}

The only way to exit the normalFunc is by returning from it, or throwing an error. If you call the function again, it will begin the execution from the top ***again***.

In contrast, a generator is a function that **can stop midway** and then continue *from where it stopped.*

Here are some other common definitions of generators —

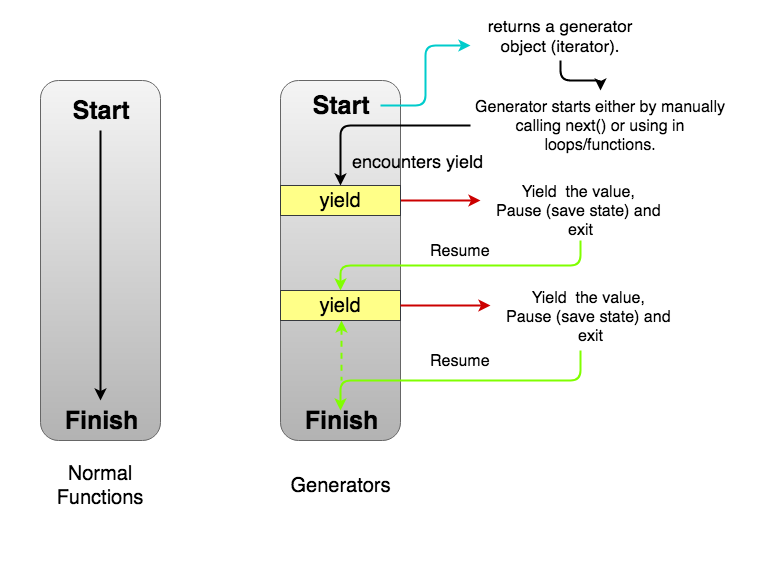
* Generators are a special class of functions that simplify the task of writing iterators.
* A generator is a function that produces a sequence of results instead of a single value, i.e you *generate*​a series of values.

In JavaScript, a generator is a function which returns an object on which you can call next(). Every invocation of next() will return an object of shape —

{   
 value: Any,  
 done: true|false  
}

The value property will contain the value. The done property is either trueor false. When the done becomes true, the generator stops and won’t generate any more values.

Here’s an illustration of the same —



Normal Functions vs Generators

Note the dashed arrow that closes the **yield-resume-yield** loop just before **Finish**in *Generators*part of the image. **There is a possibility that a generator may never finish.** We’ll see an example later.

**Creating a Generator**

Let’s see how we can create a generator in JavaScript —

**function \*** generatorFunction() { // Line 1  
 console.log('This will be executed first.');  
 **yield** 'Hello, '; // Line 2

console.log('I will be printed after the pause');   
 **yield** 'World!';  
}

const **generatorObject** = generatorFunction(); // Line 3

console.log(**generatorObject.next()**.value); // Line 4  
console.log(**generatorObject.next()**.value); // Line 5  
console.log(**generatorObject.next()**.value); // Line 6

// This will be executed first.  
// Hello,   
// I will be printed after the pause  
// World!  
// undefined

Focus on the bold parts. For creating a generator function, we use function \*syntax instead of just function. Any number of spaces can exist between the function keyword, the \*, and the function name. Since it is just a function, you can use it anywhere that a function can be used i.e inside objects, and class methods.

Inside the function body, we don’t have a return. Instead, we have another keyword yield (Line 2). It’s an operator with which a generator can pause itself. Every time a generator encounters a yield, it “returns” the value specified after it. In this case, Hello, is returned. However, we don’t say “returned” in the context of generators. We say the “the generator has ***yielded***Hello, ”.

We can also return from a generator. However, return sets the doneproperty to true after which the generator cannot generate any more values.

function \* generatorFunc() {  
 yield 'a';  
 **return** 'b'; // Generator ends here.  
 **yield 'a';** // Will never be executed.   
}

In Line 3, we create the generator object.**It seems like we are invoking** the function generatorFunction. Indeed we are! The difference is that instead of returning any value, a generator function *always*returns a generator object. The generator object is an iterator. So you can use it in for-of loops or other functions accepting an iterable.

In Line 4, we call the next() method on the generatorObject. With this call, the generator begins executing. First, it console.log the This will be executed first. Then, it encounters a yield ‘Hello, ‘. The generator yields the value as an object { value: 'Hello, ', done: false } and suspends/pauses. Now, it is waiting for the next invocation.

In Line 5, we call next() again. This time the generator wakes up and begin executing from where it left. The next line it finds is a console.log. It logs the string I will be printed after the pause. Another yield is encountered. The value is yielded as the object { value: 'World!', done: false }. We extract the value property and log it. The generator sleeps again.

In Line 6, we again invoke next(). This time there are no more lines to execute. Remember that every function implicitly returns undefined if no return statement is provided. Hence, the generator returns (instead of yielding) an object { value: undefined, done: true}. The done is set to true. This signals the end of this generator. Now, it can’t generate more values or resume again since there are no more statements to be executed.

We’ll need to make new another generator object to run the generator again.

**Uses of Generators**

There are many awesome use cases of generators. Let’s see a few of them.

**Implementing Iterables**

When you implement an iterator, you have to manually make an iterator object with a next() method. Also, you have to manually save the state. Often times, it becomes really hard to do that. Since generators are also iterables, they can be used to implement iterables without the extra boilerplate code. Let’s see a simple example.

*Problem: We want to make a custom iterable that returns This, is, and iterable.. Here’s one implementation using iterators —*

const iterableObj = {  
 [Symbol.iterator]() {  
 **let step = 0;**  
 return {  
 next() {  
 **step++;**  
 if (step === 1) {  
 **return { value: 'This', done: false};**  
 } else if (step === 2) {  
 **return { value: 'is', done: false};**  
 } else if (step === 3) {  
 **return { value: 'iterable.', done: false};**  
 }  
 return { value: '', done: true };  
 }  
 }  
 },  
}

for (const val of iterableObj) {  
 console.log(val);  
}

// This  
// is   
// iterable.

Here’s the same thing using generators —

**function \*** iterableObj() {  
 yield 'This';  
 yield 'is';  
 yield 'iterable.'  
}

for (const val of iterableObj()) {  
 console.log(val);  
}

// This  
// is   
// iterable.

You can compare both the versions. It’s true that this is some what of a contrived example. But it does illustrate the points —

* We don’t have to worry about Symbol.iterator
* We don have to implement next().
* We don’t have to manually make the return object of next() i.e { value: 'This', done: false }.
* We don’t have to save the state. In the iterator’s example, the state was saved in the variable step. It’s value defined what was output from the iterable. We had to do nothing of this sort in the generator.

**Better Async functionality**

Code using promises and callbacks such as —

function **fetchJson**(url) {  
 return fetch(url)  
 .then(request => request.text())  
 .then(text => {  
 return JSON.parse(text);  
 })  
 .catch(error => {  
 console.log(`ERROR: ${error.stack}`);  
 });  
}

can be written as (with the help of libraries such as [co.js](https://github.com/tj/co))—

const fetchJson = co.wrap(function \* (url) {  
 try {  
 let request = **yield** fetch(url);  
 let text = **yield** request.text();  
 return JSON.parse(text);  
 }  
 catch (error) {  
 console.log(`ERROR: ${error.stack}`);  
 }  
});

Some readers may have noticed that it parallels the use of async/await. That’s not a co-incidence. async/await follows a similar strategy and replaces the yield with await in cases where promises are involved. It is based on generators.

**Infinite Data Streams**

It’s possible to create generators that never end. Consider this example —

**function \*** naturalNumbers() {  
 let num = 1;  
 while (true) {  
 **yield num;**  
 num = num + 1  
 }  
}

const numbers = naturalNumbers();

console.log(numbers.next().value)  
console.log(numbers.next().value)

// 1  
// 2

We make a generator naturalNumbers. Inside the function, we have an infinite while loop. In that loop, we yield the num. When the generator yields, it is suspended. When we call next() again, the generator wakes up, continues from where it was suspended (in this case yield num) and executes till another yield is encountered or the generator finishes. Since the next statement is num = num + 1, it updates num. Then, it goes to the top of while loop. The condition is still true. It encounter the next line yield num. It yields the updated num and suspends. This continues as long you want.

**Generators as observers**

Generators can also receive values using the next(val) function. Then the generator is called an observer since it wakes up when it receives new values. In a sense, it keeps *observing* for values and acts when it gets one. You can read more about this pattern [here](http://exploringjs.com/es6/ch_generators.html#sec_generators-as-observers).

**Advantages of Generators**

**Lazy Evaluation**

As seen with **Infinite Data Streams** example, it is possible only because of lazy evaluation. Lazy Evaluation is an evaluation model which delays the evaluation of an expression until its value is needed. That is, if we don’t need the value, it won’t exist. It is **calculated**as we demand it. Let’s see an example —

**function \*** powerSeries(number, power) {  
 let base = number;  
 while(true) {  
 yield Math.pow(base, power);  
 base++;  
 }  
}

The powerSeries gives the series of the number raised to a power. For example, power series of 3 raised to 2 would be **9(3²) 16(4²) 25(5²) 36(6²) 49(7²).**When we do const powersOf2 = powerSeries(3, 2); we just create the generator object. None of the values has been computed. Now, if we call next(), 9 would be computed and retuned.

**Memory Efficient**

A direct consequence of Lazy Evaluation is that generators are memory efficient. We generate only the values that are needed. With normal functions, we needed to pre-generate all the values and keep them around in case we use them later. However, with generators, we can defer the computation till we need it.

We can create combinator functions to act on generators. Combinators are functions that combine existing iterables to create new ones.One such combinator is take. It takes first n elements of an iterable. Here’s one implementation —

function \* take(n, iter) {  
 let index = 0;  
 for (const val of iter) {  
 if (index >= n) {  
 return;  
 }  
 index = index + 1;  
 yield val;  
 }  
}

Here’s some interesting use cases of take —

take(3, ['a', 'b', 'c', 'd', 'e'])

// a b c

take(7, naturalNumbers());

// 1 2 3 4 5 6 7

take(5, powerSeries(3, 2));

// 9 16 25 36 49

Here’s an implementation of [cycled](https://github.com/sindresorhus/cycled) library (without the reversing functionality).

function \* cycled(iter) {  
 const arrOfValues = [...iter]  
 while (true) {  
 for (const val of arrOfValues) {  
 yield val  
 }  
 }  
}

console.log(...take(10, cycled(take(3, naturalNumbers()))))

// 1 2 3 1 2 3 1 2 3 1

**Caveats**

There are some points that you should remember while programming using generators.

* **Generators are one-time access only.**Once you’ve exhausted all the values, you can’t iterate over it again. To generate the values again, you need to make a new generator object.

const numbers = naturalNumbers();

console.log(...take(10, numbers)) // 1 2 3 4 5 6 7 8 9 10  
**console.log(...take(10, numbers))** // This will not give any data

* Generators do not allow random access as possible with arrays. Since the values are generated one by one, accessing a random value would lead to computation of values till that element. Hence, it’s not random access.

**Conclusion**

A lot of things are yet to be covered in generators. Things such as yield \*, return() and throw(). Generators also make [coroutines](https://en.wikipedia.org/wiki/Coroutine) possible. I’ve listed some references that you can read to gain further understanding of generators.

You can head over to Python’s [itertools](https://docs.python.org/2/library/itertools.html#itertools.chain) page, and see some of the utilities that allow working with iterators and generators. As an exercise, you can implement the utilities yourself.