**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 —

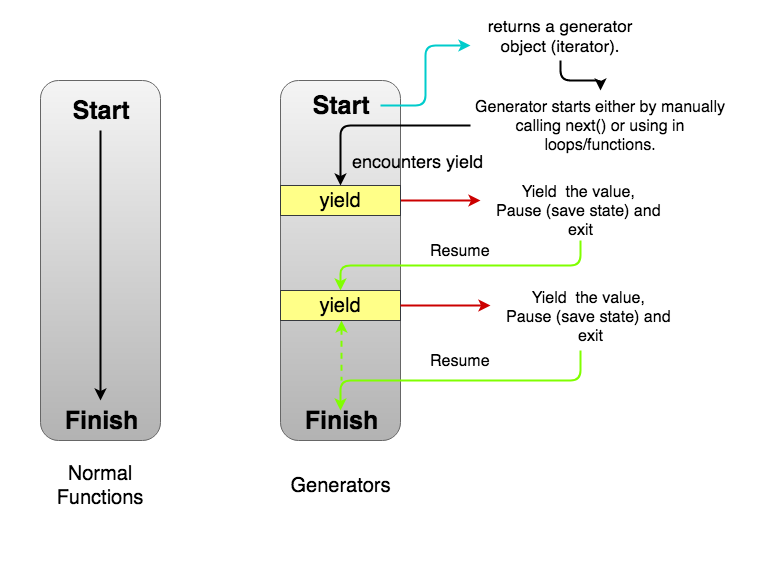
* Generators are a special class of functions that simplify the task of writing iterators.
* A generator is a function that produces a sequence of results instead of a single value, i.e you *generate*​a series of values.

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 true or 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 3console.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 done property 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.