ES6 introduces a new mechanism for traversing data: iteration. Two concepts are central to iteration:

* An iterable is a data structure that wants to make its elements accessible to the public. It does so by implementing a method whose key is Symbol.iterator. That method is a factory for iterators.
* An iterator is a pointer for traversing the elements of a data structure (think cursors in databases).

We will see that the iterable is returned from the method call to the property accessed via Symbol.iterator => this method is [Symbol.iterator]()

And the iterator is next();

Expressed as interfaces in TypeScript notation, these roles look like this:

**interface** Iterable {

[Symbol.iterator]() : Iterator;

}

**interface** Iterator {

next() : IteratorResult;

}

**interface** IteratorResult {

value: any;

done: **boolean**;

}

The following values are iterable:

• Arrays

• Strings

• Maps  
• Sets

• DOM data structures (work in progress)  
*Plain objects are not iterable*

Iterability

* Data consumers: JavaScript has language constructs that consume data. For example, for- of loops over values and the spread operator (...) inserts values into Arrays or function calls.
* Data sources: The data consumers could get their values from a variety of sources. For example, you may want to iterate over the elements of an Array, the key-value entries in a Map or the characters of a string.

So data consumers use iterable data sources to perform some action

Data consumers have a mechanism to iterate over iterable elements – this is known as an iterator

This relationship is centered around a new ES6 interface known as Iterable

Data consumers tap into the iterables that data sources implement

They are the mechanism which enable iteration

• Source: A value is considered iterable if it has a method whose key is the symbol Symbol.iterator that returns a so-called iterator. The iterator is an object that returns values via its method next(). We say: it iterates over the items (the content) of the iterable, one per method call.

• Consumption: Data consumers use the iterator to retrieve the values they are consuming.

Plain objects are not iterable

Objects aren’t iterable over properties

There are two levels at which you can iterate in JavaScript

1. The program level: iterating over properties means examining the structure of the program
2. The data level: iterating over a data structure means examining the data managed by the program

Making iteration over properties the default would mean mixing those levels, which would have two disadvantages

1. You can’t iterate over the properties of a data structure
2. Once you iterate over the properties of an object, turning that object into a data structure would break your code

If engines were to implement iterability via a method Object.prototype[Symbol.iterator](), then there would be an additional caveat: Objects creates via Object.create(null) wouldn’t be iterable, because Object.prototype is not in their prototype chain

But now we don’t really need to worry about objects not being iterable when we can do key value pairs with maps

Even if the original iterable and the iterator are not the same object, it is still occasionally useful if an iterator has the following method (which also makes it an iterable):

[Symbol.iterator]() {

return this;

}

All built-in ES6 iterators follow this pattern (via a common prototype, see the chapter on generators).

> const arr = [];

> const iterator = arr[Symbol.iterator]();

> iterator[Symbol.iterator]() === iterator true

Why is it useful if an iterator is also an iterable? for-of only works for iterables, not for iterators. Because Array iterators are iterable, you can continue an iteration in another loop:

const arr = ['a', 'b'];

const iterator = arr[Symbol.iterator]();

for (const x of iterator) {

console.log(x); // a

break;

}

// Continue with same iterator:

for (const x of iterator) {

console.log(x); // b

}

**Iterator Protocol**

Consists of iterables, iterators, and iterator results

interface Iterable {

[Symbol.iterator]() : Iterator;

}

interface Iterator {

next() : IteratorResult;

return?(value? : any) : IteratorResult;

}

interface IteratorResult {

value : any;

done : boolean;

}

Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once. After a terminating value has been yielded additional calls to next() should simply continue to return {done: true}.

Rules for next():

• As long as the iterator still has values x to produce, next() returns objects { value: x, done: false }.

• After the last value was iterated over, next() should always return an object whose property done is true.

The property done of an iterator result doesn’t have to be true or false, truthy or falsy is enough. All built-in language mechanisms let you omit done: false.

Note that each iterator in the standard library is also an iterable. Its method [Symbol.iterator]() return this, meaning that it always returns the same iterator (itself).

Closing: by calling return(), you tell the iterator that you don’t intend to call next(), anymore.

Rules for calling return():

• return() is an optional method, not all iterators have it. Iterators that do have it are called closable.

• return() should only be called if an iterator hasn’t been exhausted. For example, for-of calls return() whenever it is left “abruptly” (before it is finished). The following operations cause abrupt exits: break, continue (with a label of an outer block), return, throw.

• Documenting an iterable: provide the following information.

– Does it return fresh iterators or the same iterator each time?

– Are its iterators closable?

• Implementing an iterator:

– Clean-up activity must happen if either an iterator is exhausted or if return() is called.

\* In generators, try-finally lets you handle both in a single location.

– After an iterator was closed via return(), it should not produce any more iterator results via next().

• Using an iterator manually (versus via for-of etc.):

– Don’t forget to close the iterator via return, if – and only if – you don’t exhaust it. Getting this right can be tricky.

• Continuing to iterate over an iterator after an abrupt exit: The iterator must either be unclosable or made unclosable (e.g. via a tool class).